

Low-Latency Network Coding for Streaming Video Multicast

by

Kah Keng Tay

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science


at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

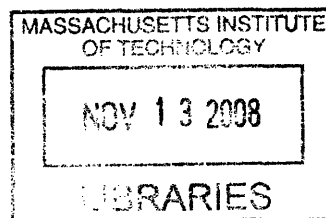
June 2008

© Massachusetts Institute of Technology 2008. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
May 23, 2008

Certified by.....

Dina Katabi
Associate Professor
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Students



ARCHIVES

Low-Latency Network Coding for Streaming Video Multicast

by

Kah Keng Tay

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Network coding has been successfully employed to increase throughput for data transfers. However, coding inherently introduces packet inter-dependencies and adds decoding delays which increase latency. This makes it difficult to apply network coding to real-time video streaming where packets have tight arrival deadlines.

This thesis presents FLOSS, a wireless protocol for streaming video multicast. At the core of FLOSS is a novel network code. This code maximizes the decoding opportunities at every receiver, and at the same time minimizes redundancy and decoding latency. Instead of sending packets plainly to a single receiver, a sender mixes in packets that are immediately beneficial to other receivers. This simple technique not only allows us to achieve the coding benefits of increased throughput, it also decreases delivery latency, unlike other network coding approaches. FLOSS performs coding over a rolling window of packets from a video flow, and determines with feedback the optimal set of packet transmissions needed to get video across in a timely and reliable manner.

A second important characteristic of FLOSS is its ability to perform both inter- and intra-flow network coding at the same time. Our technique extends easily to support multiple video streams, enabling us to effectively and transparently apply network coding and opportunistic routing to video multicast in a wireless mesh.

We devise VSSIM*, an improved video quality metric based on [46]. Our metric addresses a significant limitation of prior art and allows us to evaluate video with streaming errors like skipped and repeated frames.

We have implemented FLOSS using Click [22]. Through experiments on a 12-node testbed, we demonstrate that our protocol outperforms both a protocol that does not use network coding and one that does so naïvely. We show that the improvement in video quality comes from increased throughput, decreased latency and opportunistic receptions from our scheme.

Thesis Supervisor: Dina Katabi
Title: Associate Professor

Acknowledgments

I would like to thank God for seeing me through a difficult but fruitful year of learning and research. Big thanks to all my family and friends for their loving support and prayers, especially my parents for bringing me up and supporting me through college. This thesis would not have been possible without all of your continued encouragement for the past year.

I am thankful and indebted to Dina Katabi for taking me in as her student and research assistant, and spending time to show me how to do good research. To Sarah Lau, my wife-to-be and wonderful soulmate, thank you for always being by my side and putting up with me and my busyness for these several years.

I am especially grateful to Szymon Chachulski for his help on various matters (and his endless supply of chocolate). My thanks also go to the rest of the group for the engaging discussions and also Samir Selman for his help bootstrapping me into this project. A shout-out to my labmates Mujde, Hooria, James, my housemates Yogi, Eli, Jeremy, the TGIF gang, the Sat Night bible study, Sunday International Fellowship, my mahjong and DotA buddies, and everyone else for the memorable times at MIT.

Contents

1	Introduction	19
2	Background and Related Work	23
2.1	Network Coding	24
2.2	Rateless Codes	24
2.3	Opportunistic Routing	26
2.4	Media Streaming	26
3	A Single-Hop Video Multicast Protocol	29
3.1	Overview	29
3.1.1	Motivating Example	29
3.1.2	Two Naïve Protocols	30
3.1.3	Further Analysis	32
3.2	The FLOSS Protocol for Single-Hop Multicast	35
3.2.1	Coding Systematically	35
3.2.2	Coding Over a Rolling Window of Packets	38
3.2.3	Coding with Application Information	38
3.2.4	Protocol Description	39
3.2.5	FLOSS At Work	40
4	Generalizations to Larger Networks	43
4.1	FLOSS in Multiple-Hop Networks	43
4.1.1	Dealing with Missing Packets	44

4.1.2	Dealing with Opportunistic Receptions from Upstream	44
4.1.3	Dealing with Opportunistic Receptions from Elsewhere	45
4.2	FLOSS in Networks With Multiple Flows	45
4.2.1	Identifying Different Flows	47
4.2.2	Nodes Playing Multiple Roles	47
5	Implementation	49
5.1	Development Environment and Software	49
5.2	Primitives for Distributed Systems	49
5.2.1	Packet Sequencing	50
5.2.2	Bit Patches	50
5.2.3	Distributed State Agreement	51
5.3	Protocol Implementation	52
5.3.1	Packet Header	52
5.3.2	Control Flow	55
5.3.3	Playout Buffering	56
5.4	Coding Algorithm	56
5.4.1	Packet Selection	56
5.4.2	Modified Gauss-Jordan Elimination	57
5.4.3	Packet Expiry Policy	61
5.4.4	Incorporating Content Type Information	62
6	Video Quality Assessment	65
6.1	Available Metrics	65
6.1.1	VSSIM	66
6.1.2	Limitations of VSSIM	67
6.2	Improved Metric: VSSIM*	67
6.3	Further Optimizations	70
6.4	Comparison of VSSIM and VSSIM*	71
6.4.1	Frame Alignment	71
6.4.2	Indication of Quality	74

6.4.3	Relationship of Loss Rate and Metric Score	75
6.5	Summary	76
7	Experimental Evaluation	79
7.1	Testbed Environment	81
7.2	Performance in Single-Hop Multicast	82
7.2.1	How Do The Protocols Compare?	82
7.2.2	Where Does The Improvement Come From?	84
7.3	Performance in Multiple-Hop Multicast	87
7.3.1	1-Way Tree Topology	87
7.4	Performance in Multiple-Hop Multicast with Multiple Flows	89
7.4.1	2-way Chain Topology	89
7.4.2	3-way Conference Topology	90
8	Conclusion	93
8.1	Contributions	93
8.2	Future Work	93

List of Figures

3-1	Estimated server goodput at varying loss rates. Goodput rate decreases with number of clients but increases with the batch size. Note that the BATCH protocol with a batch size of 1 is equivalent to the BC+RETRANS protocol.	33
3-2	Estimated transmissions required per packet for BATCH at different loss rates. Increasing the batch size gives us a higher guarantee of reception, but only if we get a critical mass of transmissions across. The crossover point is the expected number of transmissions for that lost rate.	34
3-3	Estimated transmissions required per packet for slightly improved BATCH at different loss rates. We do better than the ordinary BATCH protocol, by initially sending packets uncoded.	36
3-4	Demonstration of the FLOSS protocol and coding algorithm. After the server sends 4 native packets, client A is the first to acknowledge the reception of native packets 1 and 3. After running the packet coding procedure on the client order A, B, C, the server queues an encoded packet containing native packets 1 and 2. All clients can decode and extract 1 native packet upon reception of this encoded packet.	41
4-1	Possible gains from opportunistic receptions in a multi-hop network. Coded data could get picked up by a node several hops downstream. As the node broadcasts this data to its clients, the data can get picked up by its server and neighboring nodes.	46

4-2	Multiple parties having a video conference through a wireless router. Each party is the source of one video flow, and streams it through the router to the other parties. Thus, each party acts as a server for one flow and clients of two other flows.	46
5-1	Packet header format for FLOSS. Required fields are shown shaded. .	53
5-2	Architectural overview and control flow within FLOSS. We show the control flow for the sender and receiver sides of our protocol, and that of a periodic timer at each node.	54
5-3	An example of our modified Gauss-Jordan elimination for packet decoding in FLOSS. Our algorithm allows us to depart from the notion of batches and deal with packets coded over arbitrary windows. In the above example, after the elimination is complete, we can deduce that the server just needs to send native packets 1 and 2 for the client to decode all 5 native packets.	58
5-4	Recovery of expired data using Gauss-Jordan elimination. In our example above, the dashed lines represent advancing expiry times. We can use the same algorithm to extract some data from expired packets, limiting the number of coded packets that get discarded.	61
6-1	Modeling the edit distance as a shortest path problem. Nodes in our grid network represent pairs of frames (o_x, r_y) from the original video o and received video r respectively, while edges represent transitions between two such pairs. A diagonal edge from (o_x, r_y) to (o_{x+1}, r_{y+1}) indicates that frames o_x and r_y that are aligned but possibly suffered some distortion, and is given a weight of $1 - SSIM(o_x, r_y)$. Horizontal edges correspond to frames deleted in the received video, while vertical edges correspond to frames inserted in the received video, and these are given weights of $1 - \text{delete_score}$ and $1 - \text{insert_score}$ respectively. Finding the maximum similarity in two video sequences o and r is equivalent to finding the shortest path from (o_0, r_0) to (o_m, r_n)	69

6-2	Performance of video quality metric under different UDP loss rates. We plot the individual frame scores over all frames of the video, and use horizontal lines to indicate the average over all frames. Vertical dashed lines indicate frame insertions as computed by our metric.	73
6-3	Snapshots of observed video at different loss rates. We show frames 57, 67, and 208 of the received video of our runs. These frames give a sense of possible manifestations of decoding error at various loss rates.	74
6-4	Scatter plot of video quality score against UDP loss rate. Each dot represents a streaming experiment between two nodes, plotted at their observed loss rates and computed quality. Our proposed VSSIM* shows a more defined relationship between loss rates and video quality.	77
7-1	Node locations in our testbed. The 12 nodes of our testbed are situated on a single floor of our building.	81
7-2	Comparison of streaming protocols for different bit-rate videos. Each point represents a flow from a server to one client, plotted at the loss rate and video quality experienced by that client for that flow. For low bit-rate video, FLOSS and BC+RETRANS perform better than BATCH and BC. For a video of higher bit-rate, FLOSS sustains quality to a higher loss threshold than BC+RETRANS.	83
7-3	Bar graph of marginal increase in video quality. Each bar represents, for the 3 retransmissions-based protocols, the contribution to video quality of each received packet beyond the number of packets received in the BC protocol. Thus, we are measuring the utility of each retransmission for the 3 protocols of FLOSS, BC+RETRANS and BATCH.	84

7-4	Increased throughput improves video quality. The throughput of a streaming session is throughput of video data received by the client in that session. In the cumulative distribution of throughput over all runs of 1Mb/s video, we find that FLOSS has slightly higher median throughput than BC+RETRANS, and significantly higher throughput than the other schemes. The scatter plot shows a linear relationship between throughput and video quality, which suggests that increased throughput has a direct impact on video quality improvement.	85
7-5	Analysis of latency for different protocols. The vertical bars in (b) denote average packet latency over all streaming runs. FLOSS has slightly less median and average latency compared with BC+RETRANS, while BATCH has the highest average latency. There is no clear relationship between the average packet latency and video quality.	86
7-6	Diagram of a tree topology. The arrows indicate the multicast tree between 8 nodes, formed by determining static routes from the source using the ETX metric.	88
7-7	Protocol performance on 1-way tree topology. Video quality has improved at most nodes, while the number of transmissions has a slight decrease at forwarders primarily from opportunistic receptions.	88
7-8	Diagram of 2-way chain topology. Node B sits in the middle of two opposite flows, and hence has ample opportunities for coding packets from the two flows together.	89
7-9	Protocol performance on 2-way chain topology. Both network coding and opportunistic receptions individually lead to some gain in video quality, with an average gain from 13-39%. Both of these are also responsible for decreasing the network load on the middle node by about 20%.	90
7-10	Diagram of 3-way conference topology. Each edge node is streaming video to the other 2 edge nodes via the middle router node. This middle node has opportunities for both inter- and intra-flow coding.	90

7-11 **Protocol performance on 3-way conference topology.** Again, network coding and opportunistic receptions both contribute to gains in video quality of about 5%, while the number of transmissions decreases by about 9% at the middle node. 91

List of Tables

3.1	Definitions of terms used in this thesis.	30
6.1	Comparison of regular VSSIM metric and proposed VSSIM* metric. We list for each run the overall loss rate, the number of received frames, and the computed VSSIM and VSSIM* scores.	75
7.1	A summary of experimental evaluations of the FLOSS protocol. Each experiment is described in greater detail in the respective sections listed above.	81

Chapter 1

Introduction

The demands of media streaming are increasing as multimedia applications become more common and use higher bit rates. At the same time, the physical constraints of the wireless channel are being stretched as more devices share the same frequency band. As such, there is an important need to extract as much throughput possible from the available bandwidth, so that multiple devices can share the medium and still be able to communicate effectively.

This thesis presents FLOSS (Flow Synthesis & Separation), a network multicast protocol that transparently allows network coding to be applied to real-time video streaming in a wireless mesh network. Traditionally, network coding has been used to increase throughput and reliability in wireless data transfer by coding a batch of packets together and broadcasting a certain number of encoded packets. This, however, introduces additional packet dependencies for decoding, and causes latency to increase. As such, it has been difficult to apply network coding to the problem of streaming real-time media reliably, since these packets have tight delivery deadlines. A naïve application of network coding to media streaming could, as a result of high latency, deliver poorer quality video compared to a protocol that does not use network coding.

In contrast, FLOSS uses a novel systematic network code that mixes packets together to minimize decoding latency. In essence, for each packet that is missing at a receiver and needs to be delivered, the sender codes together with it some of the

packets that the receiver already has but may be missing at other receivers. This simple scheme actually allows us to perform both inter-flow and intra-flow coding of packets effectively, and allows the mesh network to increase its wireless throughput without additional latency caused by waiting for packet dependencies to arrive.

FLOSS provides the following concrete benefits:

- **Reduced latency.** Packets are delivered with lower latency compared to schemes without network coding or naïve network coding.
- **Data compression.** Our use of coding enables us to compress data in the network, by allowing us to send more data with the same number of transmissions.

To realize these benefits, FLOSS uses two novel techniques:

- FLOSS uses application-level information about content and expiry times of data packets, and identifies the window of packets that are worth transmitting given their time to expiry.
- Using feedback, FLOSS determines the minimal set of packets any receiver still requires. It then uses a systematic network code to code, together with these packets, other packets the receiver already has. These packets are chosen in a way that maximizes the benefit to other receivers, while keeping decoding latency to a minimum.

In FLOSS, nodes use a multicast tree to deliver media through a wireless mesh network. Each node requests data from its parent node in the multicast tree, by listing the packets it has already received. Parents respond to their children by queueing for transmission packets that have not yet been received, much like a typical multicast scheme. However, a key difference is that, in FLOSS, the parent also uses a systematic network code to mix with each transmission other packets that have already been received by the child.

When compared with a multicast scheme using traditional retransmissions, the coding has the same effect on the intended recipient of the retransmission, since the child node is able to easily extract a new packet by subtracting away the packets that

it already has. However, multiple children can be satisfied with a single coded packet though they might be missing different data. FLOSS allows receivers to extract the maximum benefit afforded by this simple scheme, by providing as many opportunities for decoding as possible. Even if some data of a coded reception has reached its deadline and is now useless, FLOSS prevents the entire set of coded receptions with that data from becoming useless. Instead, FLOSS tries its best to eliminate expired data from those packets, and save what remains for decoding at a later time.

In this thesis, we begin with a study of the background and related work in Chapter 2. We next describe the mechanisms of our protocol, starting in Chapter 3 with a simple scenario of a single source sending media via multicast to several recipients a single hop away. We then extend and generalize the same protocol to multiple-hop multicast networks and networks with multiple flows in Chapter 4. We consider implementation details in Chapter 5, and describe our proposed method for video quality assessment in Chapter 6. The experimental evaluation of our protocol on a 12-node testbed is presented in Chapter 7, and we conclude in Chapter 8.

Chapter 2

Background and Related Work

Our work sits in the intersection of the broad areas of network coding, opportunistic routing and media streaming.

Network coding is a technique that successfully increases throughput and reliability by mixing together data from different sources and broadcasting the coded data. Pioneered by Ahlswede et al. [1], network coding has seen universal application in areas of data transfer [19, 14], routing [31, 8], security [20], as well as error recovery and failure resilience [16]. Linear codes [24, 21] and random linear codes [16] have been shown to be sufficient to achieve capacity in a multicast network and are useful for improving robustness.

Multimedia data, however, differs from regular data in two fundamental ways. Firstly, data pieces have an associated deadline, such as the display time for a video packet. Secondly, there are different levels of importance for different pieces of data. As such, it might be better to focus on more important data and ignore data which is not as important. Given these differences, different network codes have to be used for media applications, because of the differing requirements of reliability and timeliness.

We build on the ideas from these fields, but our scheme has a unique design and exhibits several key differences. Our design uses a novel low-latency network code to perform streaming video multicast. We are the first to utilize network coding for both inter-flow and intra-flow coding simultaneously for the benefits of increased throughput. In addition, our protocol utilizes application-level information such as

delivery deadlines and content type in packets to deliver video of better quality. Finally, FLOSS is the first streaming video multicast protocol incorporating network coding that has been implemented and evaluated on a testbed of commodity hardware, with streaming of actual video done using the popular open-source software VLC by VideoLAN [40].

2.1 Network Coding

Prior work in network coding has chosen to focus on coding within a single multicast flow [8, 24, 21, 16, 32, 48, 38], or coding between multiple unicast flows [19, 35]. Two prior practical implementations of network coding for data transfer are COPE [19], where nodes code together packets of unicast flows bound for different nexthops, and MORE [8], where they code together packets from the same flow. FLOSS builds upon the packet coding algorithm of COPE, but with an important difference: FLOSS will also code together packets from within the same flow. Thus, FLOSS uses a network code that does inter-flow and intra-flow coding concurrently, and is capable of supporting both unicast and multicast traffic of multiple flows in the same network simultaneously. This makes FLOSS applicable to a greater variety of wireless mesh deployments. At the same time, we depart from the notion of a batch in MORE. By coding in batches, nodes in our network have to wait to collect a certain number of packets before being able to decode any of them. If the data expires by that time, all the used bandwidth is wasted, and we get poor quality of video. Instead, FLOSS codes incrementally, trying to allow nodes to extract a useful packet immediately from each transmission.

2.2 Rateless Codes

FLOSS also builds upon work in rateless codes. LT (or Luby Transform) codes [26] are the first realization of a rateless erasure code that can be used for a digital fountain [7, 29]. LT codes are capable of producing practically an unlimited number of encoded

packets for sending some packetized data in a one-way multicast, and a receiver only needs to receive a number of packets slightly greater than the original number in order to be able to successfully decode and recover all the packets. Raptor codes [37] uses LT codes with an outer code for correcting erasures of known rate, and has been shown to be better in practice than pure LT codes.

These codes are rateless because a destination only needs to receive a certain number of packets in any way, independent of the channel rate, for it to be able to successfully decode all packets with high probability. Thus a sender can serve a heterogenous set of receivers at the same time with the same transmissions.

FLOSS achieves reliability differently, through the use of feedback. Nodes inform their neighbors of the coded receptions they have received in an efficient way using sequence numbers, and thus every node can retransmit missing packets. This enables us to gain two advantages which are especially significant in the application of real-time video streaming. First, the video source does not need the entire data to be available beforehand for pre-coding; rather, it can code packets as they become available. Secondly, nodes in our network can expect to receive some immediate benefit from each transmission. In other words, packet erasures can be quickly identified and action taken to address it before that packet has reached its deadline. Unlike rateless codes, which are designed to give high guarantees that a receiver can decode a whole batch of data after receiving a certain number of packets, our code tries to maximize the immediate benefit to each receiver at any time. This is more similar in spirit to growth codes [18], which codes packets to maximize the amount of information that can be recovered by a sink node at any time. [11] expands on this coding technique to perform unequal error protection of different types of video frames. However, FLOSS is fundamentally different, because it uses a mixture of application feedback and feedback from nodes to ensure packets are not decoded after their deadline.

2.3 Opportunistic Routing

FLOSS also builds upon prior work in opportunistic routing. ExOR [3] coordinates the forwarding of packets by nodes in a mesh. The source first broadcasts a set of packets, and waits for an acknowledgment from the destination. Then, nodes follow a schedule to forward packets they have heard that have not yet been received by the destination nor been broadcasted. Nodes go in order of increasing values of ETX [10] from the destination, essentially beginning with nodes that are closest to the destination and gradually fanning out. This effectively allows a packet to jump several hops toward the destination with one transmission, and saves transmissions by having nodes forward what they have heard as opposed to specifying a destination for each transmission. However, imposing a global schedule prevents effective spatial reuse. MORE [8] takes a different approach by using random network coding. Nodes send random linear combinations of packets they have heard, in a distributed manner. MORE uses the notion of transfer credit to determine how often a node can transmit, based on its proximity to the destination and the information content it has received from other nodes. This decentralized protocol allows information to propagate opportunistically through the mesh towards the destination. FLOSS takes an approach that combines ideas from these two pieces of work. Forwarding nodes in FLOSS wait a short while for downstream destinations to globally acknowledge any packets they have received or decoded, before sending new data. This allows us to gain from opportunistic receptions that might have traversed multiple hops. In addition, every node listens opportunistically for any data that might be broadcasted, coded or otherwise, and sends out linearly coded combinations of packets they have decoded.

2.4 Media Streaming

NCVD [35] uses the idea of COPE to apply coding to unicast video traffic in a wireless mesh, while CodeCast [32] looks at intra-flow coding for multicast video

traffic in varying sizes of batches. Both of these achieve better performance for video by using information from the video stream. NCVD chooses packets to code based their pre-computed contribution to video quality, while CodeCast determines the ideal batch size based on the delay constraints of the video and knowledge of the channel conditions. FLOSS, in contrast, has no need to determine a batch size or estimate the channel. Instead, FLOSS uses timing information available from the video stream to determine the deadline of each packet, and chooses packets from the window of unexpired packets to code. FLOSS also has the capability to use the content type of data packets to determine their relative importance on-the-fly, and this information is used to improve video quality.

Raptor codes have been applied to streaming video [42, 41] where nodes in a multihop network perform re-encoding and forwarding of packets to exploit network diversity. RPB and RBS [28] looks at the digital fountain approach to video, while Sliding-Fountain [5] uses the idea of a window of packets to better accommodate the delay and buffer constraints of video streaming. However, these fundamentally still suffer from the same problem of different reliability guarantees for application to video. Also, these codes could potentially still be coding data that has already passed its deadline, or data that is too far in the future for which the client has no buffer space [5], especially for variable bit-rate video. FLOSS, on the other hand, will stop coding packets if these no longer can add value to receiving nodes.

Chapter 3

A Single-Hop Video Multicast Protocol

3.1 Overview

In this chapter, we study the scenario of a single-source single-hop video multicast network. We start by describing as our baseline a simple multicast protocol and highlight its limitations. We then explain how a simple application of network coding could lead to improved network throughput. However, we also demonstrate that the resultant dependencies from network coding could in fact lead to wasted bandwidth. We then describe our proposed network protocol FLOSS, and explain why this protocol allows us to achieve the above benefits of network coding without the disadvantages of increased latency. We also look at various improvements our protocol can have by examining the content of the video, in particular, the expiry deadlines as well as the relative importance of each type of content.

3.1.1 Motivating Example

Imagine a lecture by a popular professor. There are not enough seats in the lecture theatre, so the lecture is captured on camera and broadcast live at various access points (APs) around the campus, allowing students to watch it on their laptops.

Term	Definition
Server or upstream node	A parent node in the multicast tree.
Client or downstream node	A child node in the multicast tree.
Native packet	A raw data packet from the video application.
Coded packet	A linear combination of several raw packets, with coefficients randomly chosen over a finite field.
Expired packet	A packet that has passed its display deadline.
Innovative packet	A coded packet that contains at least one native packet that is new to the client.
Packet innovation	The number of packets mixed together in a coded packet that are new to the client.
Data portion	The part of the packet containing coded data.
Response portion	The part of the packet containing protocol metadata.
Packet window	The window of unexpired packets from a source.

Table 3.1: **Definitions of terms used in this thesis.**

In this example, each AP needs to broadcast video data to its clients via the wireless channel. If we model the wireless channel as a packet erasure channel, each recipient might experience lost video packets that would affect video quality. We would like to use feedback and retransmissions to address such losses and improve reliability. For this thesis, we shall call an AP the server of a video flow, while the receiving nodes are the clients of this flow in the multicast network. We summarize our thesis terminology in Table 3.1.

3.1.2 Two Naïve Protocols

We first describe a simple protocol BC+RETRANS that does not use network coding, and instead simply broadcasts packets to its multicast group. If any packets get lost, the protocol retransmits the missing packets using broadcast as well. If we look at the high-level ideas and skip the implementation details for now, the protocol for BC+RETRANS could be described as such:

- The server broadcasts native packets as they are received from the video encoder.
- Client nodes periodically inform the server of the packets they have received through response packets.

- The server looks at each response as it is received, plus the packets currently queued for transmission, and determines the packets that will still be missing at the client. It then enqueues these packets for retransmission via broadcast.

The problem with the above protocol is that if there are multiple clients, and each client is missing a different packet, the server is burdened with retransmitting many packets, one for each client. Each retransmission can only benefit one client, and thus is not an efficient use of bandwidth.

Contrast this with a scheme BATCH that uses network coding in the following way:

- The server splits consecutive native packets from the video encoder into batches of a certain size, and broadcasts coded packets from every batch.
- Client nodes periodically inform the server of the number of packets they have received from each batch.
- The server looks at the responses and determines for each batch the maximum number of missing packets over all clients. It then queues for broadcast that number more coded packets for each batch.

The BATCH protocol can outperform BC+RETRANS because each coded transmission from the server can benefit multiple clients at the same time. Once a client has received as many packets as the batch size, it can simply invert the matrix of received coefficients and decode all the packets in the batch.

This protocol, however, has a big disadvantage. Clients are unable to decode any packet until it has enough packets to decode the whole batch at once. This additional dependency between packets results in a risky all-or-nothing property and could cause high latency. Also, when dealing with real-time video in a congested or highly lossy channel, this problem is further aggravated. If a client has not received enough packets by the time of its deadline, the server has two choices to deal with the problem. It can choose to extend the playout deadline and keep transmitting packets from that batch until all clients have received the whole batch. This has a negative effect of playback stalling. With real-time video, this is impractical as the deadline

cannot be repeatedly extended.

Alternatively, the server could choose to ignore this batch and move on to the next one. Since a whole batch of packets was not decoded by the client, the client experiences video of very poor quality during that time. Additionally, there is the possibility that bandwidth was wasted because several transmissions did not ultimately result in delivered packets to any client.

Either way, there seems to be possibly severe consequences to the simple application of network coding for real-time video. In the next section, we study this in greater detail.

3.1.3 Further Analysis

Let us compare the performances of these two protocols and show our above analysis quantitatively and probabilistically. We use a simplified model of the wireless channel as a packet erasure channel with a Bernoulli probability of loss. We note that the BC+RETRANS protocol is identical to the BATCH protocol with a fixed batch size of 1. Thus, it suffices to compare the performance of the BATCH protocol for varying batch sizes. We look at two different metrics: expected goodput rate and average transmission latency.

For each client c , let X_c be a random variable for the number of server transmissions needed before the client can decode the whole batch of size S . The server thus needs to send $\max_c \{X_c\}$ coded transmissions for each batch in order for all clients to successfully decode that batch. Assuming we are dealing with ordinary data transfers and unlimited network bandwidth, what is the goodput of such a protocol in the long run? We compute the expected number of transmissions needed and take its inverse to get the goodput rate as a function of the loss rate. In Figure 3-1, we plot these values for varying client numbers and batch size. We notice that as we increase the number of clients, the goodput rate decreases, as it is harder to ensure reception at multiple clients. However, as we increase the batch size, we get closer to achieving the theoretical maximum goodput possible for our channel conditions. Thus, coding in batches has clear benefits when there are multiple clients.

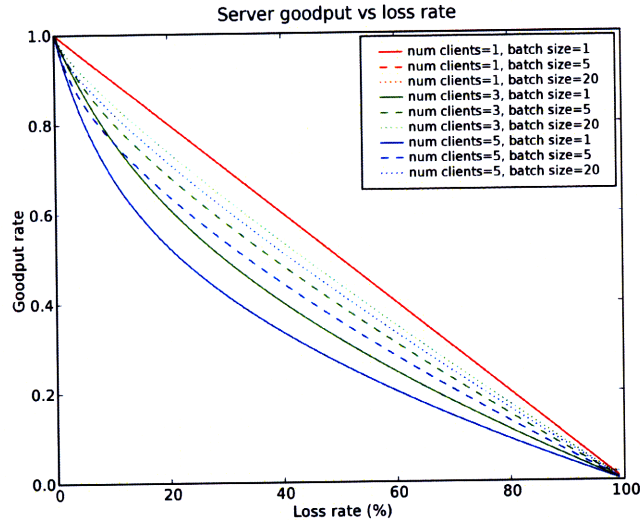
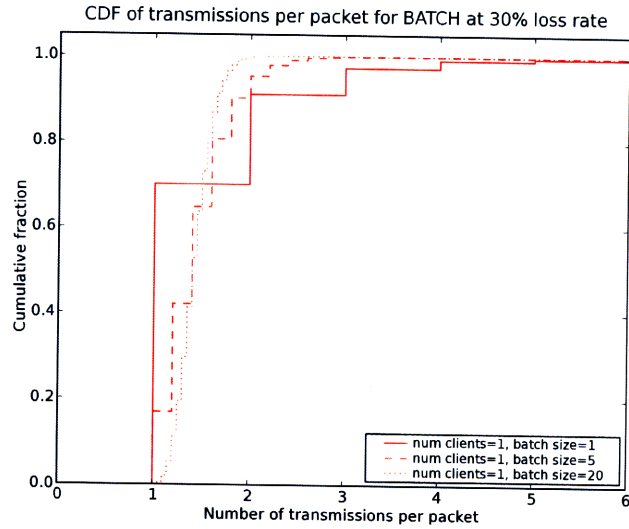


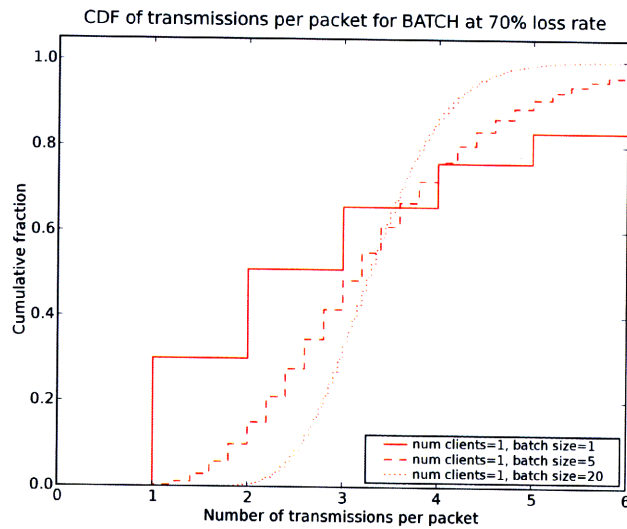
Figure 3-1: **Estimated server goodput at varying loss rates.** Goodput rate decreases with number of clients but increases with the batch size. Note that the BATCH protocol with a batch size of 1 is equivalent to the BC+RETRANS protocol.

However, we also need to consider the possible delays introduced by network coding. We do this by looking at the random variable $N = \frac{X_r}{S}$, the number of transmissions needed to get one packet of data across, and determining the cumulative distribution of its probability distribution. For simplicity, we plot this distribution in Figure 3-2 for a single client under Bernoulli loss rates of 30% and 70%. As expected, the larger the batch size, the less likely delivery can be done with an average of close to one transmission for each packet in the batch, because of the all-or-nothing property. However, there is a higher guarantee of receiving all the packets for a given number of transmissions, provided this number is sufficiently high, because random network coding evenly distributes losses over all packets in the batch.

Ideally, we would like a protocol that performs coding opportunistically and cautiously. We code packets to maximize throughput while minimizing decoding latency, to avoid the sharing of fate by all packets in a batch.



(a) Loss rate of 30%



(b) Loss rate of 70%

Figure 3-2: **Estimated transmissions required per packet for BATCH at different loss rates.** Increasing the batch size gives us a higher guarantee of reception, but only if we get a critical mass of transmissions across. The crossover point is the expected number of transmissions for that lost rate.

3.2 The FLOSS Protocol for Single-Hop Multicast

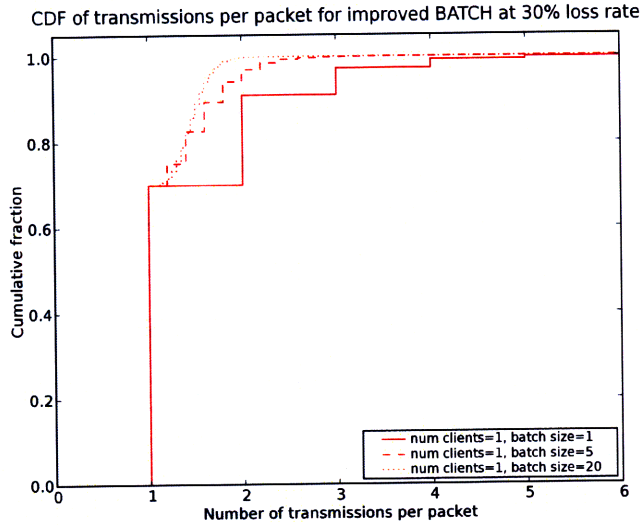
Our proposed protocol FLOSS is one that allows us to achieve these ideals. In a lossy channel with multiple client nodes, FLOSS is able to leverage network coding to bring about increased throughput. Yet, it does not perform any poorer than BC+RETRANS at its worst. In other words, we are able to get the benefits of network coding without the problems of increased latency and fate-sharing. In fact, on average, the increased throughput from network coding translates into reduced latency when compared with BC+RETRANS.

Our protocol relies on the following three ideas: 1) the use of a systematic code that puts a cap on innovation, 2) the adoption of a rolling window of packets as a batch, and 3) the incorporation of application information in the code. We describe each of these ideas in detail.

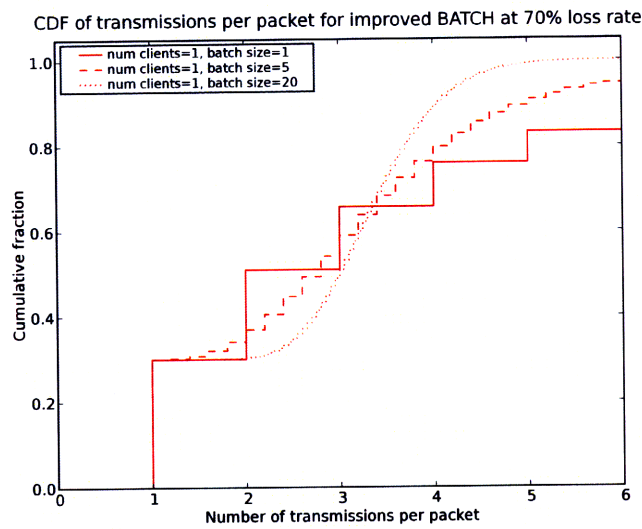
3.2.1 Coding Systematically

In our analysis of BATCH in Section 3.1.3, we saw the advantages and disadvantages of network coding in video multicast. We build on this naïve protocol and develop a systematic code that gives us better performance than BATCH.

Consider the following simple improvement to the BATCH protocol. Instead of sending coded packets from the start, the server first sends all individual native packets in the batch exactly once each. Thereafter, the server will send coded packets for any further transmissions from this batch. This improvement does not add complexity to the decoding of packets nor does it deprive the performance of BATCH. These initial transmissions are still independent linear combinations of packets in the batch, albeit with the coefficient vectors having just a single non-zero coefficient each. Sending packets this way, however, allows some packets of the batch to be immediately usable by the clients without the packets having to share the fate of the whole batch. This improvement decreases decoding latency compared to just plain BATCH.



(a) Loss rate of 30%



(b) Loss rate of 70%

Figure 3-3: **Estimated transmissions required per packet for slightly improved BATCH at different loss rates.** We do better than the ordinary BATCH protocol, by initially sending packets uncoded.

The plot of the CDF for this simple improvement will look like Figure 3-3. Note the distribution is always at least as good as, if not better, than the distribution

for BATCH. Additionally, we note that this time we have identical performance to BC+RETRANS for the first round of packet transmissions, as indicated by the jump in CDF at $N = 1$. But after the first round, using network coding for the remaining unreceived packets of the batch brings advantages over BC+RETRANS, because as before, we are able to distribute losses evenly over all packets in the batch.

The realization from this simple study is, therefore, that initially there is no advantage to sending multiple innovative packets coded together when uncoded innovative packets can get the same value across but has the added benefit of graceful degradation in the presence of losses. We can generalize this observation further for the application of network coding to video multicast: while mixing packets using randomly constructed linear codes enables us to achieve the multicast capacity, the added latency might outweigh the benefits of higher throughput. Hence, our systematic code seeks to limit the amount of innovation a packet can contain with respect to each client. This lets us strive for the maximum benefit that network coding can give us without increasing the latency. With a cap to the innovation, our target recipients should be able to decode the packet immediately.

Thus, in our systematic code, when a client sends a response indicating the packets that it has received, the server will retransmit any missing ones. However, with each enqueued packet to a client, unlike BATCH which codes with it all other packets of the batch, our code will only linearly encode other packets that have already been received by that client. This caps the innovation to 1 new packet for that client.

FLOSS chooses packets to encode such that other clients have the best chance to gain new data from this coded transmission. At first, the coded transmission is just a single packet, decided based on the client response. FLOSS considers the remaining clients one at a time in a random order for fairness. Then, for each new client considered, FLOSS checks if the current coded packet could possibly give new information to that client. In other words, it checks if the coded packet is innovative. If so, it skips to the next client. Otherwise, we add a packet to make the coded packet innovative for this client, while ensuring that previous clients still can immediately decode it and extract one unit of data. This helps to increase throughput while

ensuring that latency remains low.

3.2.2 Coding Over a Rolling Window of Packets

Our next idea is to extend the above systematic code to a rolling window of packets, instead of applying the code to batches of fixed size or batches with predetermined boundaries. Batches are used traditionally [8, 32] to facilitate decoding; one simply needs to invert a matrix of code coefficients to obtain the decoded packets. We depart from using batches and embrace a rolling window for two reasons. Firstly, when choosing packets for coding, the batch boundaries impose artificial limits on the set of packet candidates. Secondly, such partitioning is inefficient. Partially recovered data from a batch cannot be applied to the decoding of a later batch. At the same time, excess packet receptions for a batch is redundant and also unhelpful towards the decoding of any other batch.

With a rolling window, we can set our window to as large a size as necessary to maximize our packet selection ability. With our systematic code's innovation cap, the unbounded window size is not a problem for decoding. We make use of a modified Gauss-Jordan elimination algorithm to handle packets that were coded arbitrarily over any window of packets. This algorithm is described in detail in Section 5.4.2.

3.2.3 Coding with Application Information

Our third idea is to have our protocol exploit information available from the application to perform better coding. Incorporating such information into our scheme allows us to address the differences between video and conventional data. We describe two pieces of information that we use in our scheme.

(a) Packet Deadlines. Video packets have playback deadlines, and if these have passed, the packets no longer have any value to any client. At the same time, any packet that has not yet expired is potentially useful to a client, and hence should always be considered for coding as far as possible. Thus, with the above ability to

code over a rolling window of packets, we can set our window to contain all currently unexpired packets. This gives our packet selection procedure the best and most relevant set of packets to code over.

(b) Packet Content Type. Video packets have varying levels of importance. For example, I-frames are the key frames in a video sequence, while P- and B- frames are not as important. Our protocol can give preference to picking I-frame packets over other packets during packet selection phase of coding.

Furthermore, we note that the P- frames depend on I-frames, while B- frames depend on P- and/or I-frames. Our protocol can selectively code packets to emulate these dependencies, by only picking packets once the packets of their parent frame has been completely received. In this thesis, however, we do not evaluate the impact of using this information.

3.2.4 Protocol Description

The protocol can be summarized as such:

- The server stores packets as they are received from the video encoder. These packets get added to the packet window, while expired packets are removed from the window.
- Client nodes periodically inform the server of the packets they have received. This interval is their response period.
- The server looks at each response as it is received, plus the packets currently queued for transmission, and determines the packets that will still be missing at the client. It then adds these packets to its queue for retransmission. For each packet, the server also runs the code selection algorithm to code together with it several other packets.
- The server also has a timer that fires periodically, but gets postponed every time it receives a response from some client. When the timer does trigger, the server sends out unencoded packets that has not been sent before.

The timer is necessary because our scheme is essentially pull-based. If client pulls do not get received at the server because of congestion or losses, the server might end up sitting on packets that are approaching their deadline. Thus, the timer is scheduled to fire every response period. In that time, clients are expected to respond at least once. From the response, the server can proceed to send these new packets.

3.2.5 FLOSS At Work

We give a demonstration of the FLOSS protocol at work. Figure 3-4 shows a source node sending data to 3 clients. So far, 4 native packets have been transmitted, with a subset having been received by each client. Client A is the first to send a response to the server, stating that he has packets 1 and 3. The server begins to prepare a coded packet for transmission. The first missing packet is packet 2, and so this is added to the coded packet. Next, the server looks at the other clients in a random order, say B and then C. For client B, packet 2 is not innovative, thus the server adds a random packet from what B is missing that A already has. In this case, the only possibility is packet 1, since B already has packet 3, so packet 1 gets coded in. Then, we proceed to client C. Since the coded packet now contains packets 1 and 2, and 2 is innovative to C, no extra packet gets coded with this packet. Then, any of the three clients that receives this coded packet can immediately decode one native packet.

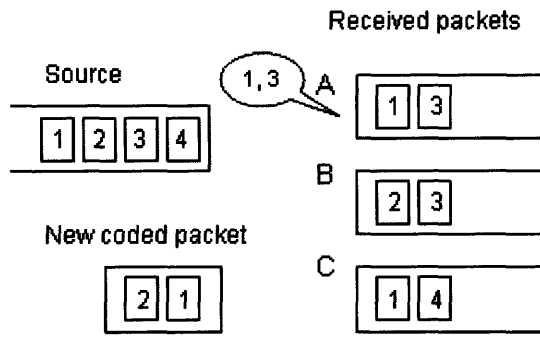


Figure 3-4: **Demonstration of the FLOSS protocol and coding algorithm.** After the server sends 4 native packets, client A is the first to acknowledge the reception of native packets 1 and 3. After running the packet coding procedure on the client order A, B, C, the server queues an encoded packet containing native packets 1 and 2. All clients can decode and extract 1 native packet upon reception of this encoded packet.

Chapter 4

Generalizations to Larger Networks

So far, we have considered the FLOSS protocol in simple single-hop multicast networks, with a single node serving video to multiple client nodes. In this chapter, we extend the protocol to networks with multiple hops for a single flow and multiple flows.

4.1 FLOSS in Multiple-Hop Networks

We consider networks with multiple hops and a single node serving video data. FLOSS assumes an existing multicast graph in the form of a tree has been constructed over the nodes, and that this topology is known by all nodes. We list possible techniques for constructing this graph in Section 8.2. We use a tree as opposed to a general directed acyclic graph so that each node other than the source has exactly one parent, and this parent will be the only node responsible for transmitting coded packets to satisfy that client. This prevents possible redundancy if there were multiple parents that happen to send the same data to their common child. Avoiding such redundancy is possible with more synchronization between parents, but this is an issue that we do not tackle here.

For FLOSS to support multiple hops, what needs to change? Since there are

nodes which are multiple hops away from the source, the intermediate hops have to act as forwarders of flow data. Each such node acts like a server to its children, and performs the same coding algorithm that the source does for its children. At the same time, these nodes still have to respond to their parent to acknowledge data they have received. Thus, in the multi-hop case, each node now has to play both the roles of a server and client. Formally, a node has to act like a server if it has at least one client or downstream node in the multicast tree, and conversely, a node has to act like a client if it has a parent node.

However, there are three key modifications that we need to make to our protocol. We describe these below.

4.1.1 Dealing with Missing Packets

The forwarding node might not have received a complete set of packets from its server yet. In this case, during packet coding, the node omits packets that has not yet been received. Instead, it will choose among packets that it has in its store that has not yet expired.

4.1.2 Dealing with Opportunistic Receptions from Upstream

When an upstream node transmits a coded packet, it might be opportunistically received by nodes multiple hops downstream. For an intermediate node to begin forwarding any decoded native packets immediately could lead to wasted bandwidth, since all its child nodes might have already received those packets from an opportunistic reception. Thus, upon successful decoding of a native packet, a timestamp is given to the packet. They will only be considered for sending to clients after a whole response period has passed. In that time, the node should expect to hear some clients respond, since they respond at least once every period, and can determine if these packets have been received.

4.1.3 Dealing with Opportunistic Receptions from Elsewhere

One final feature we add is extraneous coding in the packet coding algorithm. After the node has obtained a coded packet from the regular coding algorithm, we add to it several random packets from the set of packets that have been received by all clients. This does not change anything for the clients, because they already have the extra packets, and can still decode the coded packet.

Why then is this helpful? This gain comes from possible opportunistic receptions. This node could have obtained some decoded packets by picking up coded transmissions from another branch in the multicast tree. These new decoded packets might not have been received yet by the server of this node. By coding some extra packets that do not affect the clients (except in terms of decoding complexity), we could potentially have the client benefit the server or a node benefit another in a neighboring branch. Figure 4-1 shows 3 possible scenarios where opportunistic receptions of extraneously coded packets could help. From left to right, we have first, a server whose transmission traversed multiple hops. Second, we have a downstream client that broadcasted content which it obtained opportunistically, thereby allowing its server to get it. And finally, we have the case of a node whose packet gets received by a node in a neighboring branch.

To limit the complexity in decoding, we have a tunable parameter we call the `code.width` which limits the number of packets we may add in total to an originally uncoded packet.

4.2 FLOSS in Networks With Multiple Flows

A final cumulative generalization we make in this thesis is the support of multiple flows within the multiple-hop mesh network. Adding this allows our framework to become one which does both inter- and intra-flow coding of packets travelling within the mesh.

In a multiple-flow network, there are multiple source nodes each serving one or more flows of video data. We keep the assumption from the multi-hop case, that there

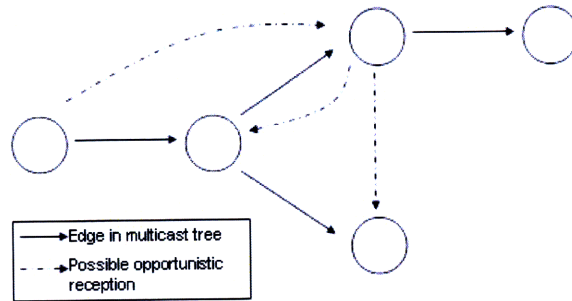


Figure 4-1: **Possible gains from opportunistic receptions in a multi-hop network.** Coded data could get picked up by a node several hops downstream. As the node broadcasts this data to its clients, the data can get picked up by its server and neighboring nodes.

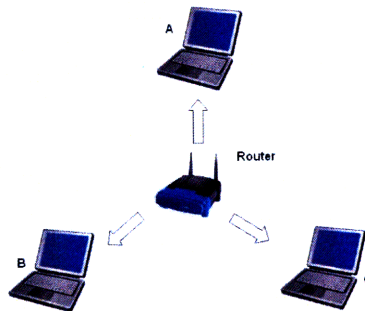


Figure 4-2: **Multiple parties having a video conference through a wireless router.** Each party is the source of one video flow, and streams it through the router to the other parties. Thus, each party acts as a server for one flow and clients of two other flows.

exists a multicast tree over all nodes that is known by all nodes. Furthermore, we extend this assumption for multiple flows, and assume that there is a multicast tree for each flow, and each tree contains all the nodes in the network. In other words, every node is a node in the multicast tree for every flow, and there are no nodes that are not participating in a particular flow. This is not a big assumption, and is made so that we do not have to deal with the issue of resource allocation here – every flow is equally important to every node. It is possible to envision a scenario where every node in the wireless mesh is a participant in video streaming sessions with everyone else. One possibility is an n -way live video conference, where audio and video feeds from all n parties are streamed to all other parties at the same time through a mesh network. While it might seem pointless to do such teleconferencing over wireless,

we argue that as improvements in wireless technology allow communication ranges to increase to tens of miles and beyond, the utility of such a system would increase. Figure 4-2 shows an example of such a network, with only a single wireless router acting as a forwarding node for all flows.

Again, we consider the changes that are needed for FLOSS to accommodate multiple flows.

4.2.1 Identifying Different Flows

Previously, there was only the notion of a single flow in the network. Hence, all nodes can assume that data packets they are handling all belong to the same flow. However, this is no longer the case. To properly differentiate the flows of packets coming from the video streaming application, we use the port number as an identifier for the flow, as opposed to the IP address of the flow source. Logically, this makes sense. When a server starts a multicast session with several clients in the network, it selects a multicast address and a port. Clients then need to start their video players and listen on a port for that streaming session. At the same time, it is possible for a node to be the source of multiple flows at the same time. It just needs to use a different port for each flow.

With an identifier for each flow, all the state that used to be kept within each node now has to be duplicated for each flow. Such state includes packet timestamps for expiry purposes, packet reception impressions for clients and servers, as well as code coefficient vectors in coded packets.

4.2.2 Nodes Playing Multiple Roles

Every node is now possibly a server and client for multiple flows at the same time. As such, any packet that it receives could come from one of its clients trying to respond to it for some flow, or it could be coming from one of its servers for some flow, or it could be both at the same time or, in the case of overheard packets, none of these. Thus, for each received packet, the node needs to check whether the sender is a server

and/or a client for any of its flows, and process the packet accordingly.

Similarly, when sending data, a node needs to act as a server to potentially clients from different flows. When sending responses to servers, a node will need to do so as a client of multiple flows, and acknowledge packet receptions for each of its servers.

In the small video conference scenario in Figure 4-2, each edge node is a server of their own video flow, with the router as their client for that flow. But at the same time, each edge node is also the client of the router for the 2 other video flows.

Chapter 5

Implementation

5.1 Development Environment and Software

We use the modular software router Click [22] to implement our network protocol, and we modify VideoLAN's VLC media player [40] to support our streaming solution, and for display and recording of video.

Our code runs as a user-space daemon, sending and receiving raw 802.11 frames using a libpcap-like interface to the wireless device. This exposes a new network interface to applications, and can be treated like any other network device such as `eth0`. We use VLC as a user application to stream video to a broadcast address and port on this interface, using the real-time transport protocol (RTP [34]). In our implementation, the 16-bit port number is used as an identifier for the flow. Clients of a specific streaming session have to instruct their players to listen to this port. It is possible for a single node to stream multiple video flows at the same time, using different port numbers.

5.2 Primitives for Distributed Systems

Before describing the protocol itself, we describe three implementation primitives we use in our protocol.

5.2.1 Packet Sequencing

Each native packet from a flow source is given a 32-bit packet identifier and a timestamp. We use the identifier for listing packets in a coded transmission, and for determining if packets lie within the unexpired window of packets.

Additionally, each (possibly coded) packet sent by a node is given a 32-bit sequence number. This number uniquely identifies the packet that was sent, plus any data that it contained. It is independent of the actual flow data that was sent and the number of flows a node is involved with. Clients of this node use this sequence number to acknowledge coded packet receptions using bit patches, which are described in the next section. At the same time, this sequence number is used by both clients and servers of the node to perform distributed state agreement, as described in Section 5.2.3.

5.2.2 Bit Patches

Unlike COPE, where the coding window is limited to packet receptions over a short period (0.5s by default), our coding window is set to match the window of unexpired packets. We could have more packets to acknowledge than we can accommodate in a single bitmap. Yet, the packets we wish to acknowledge could be sparsely distributed within the window.

Growth codes [18] weigh two different compression schemes for their header, and pick the best one. We extend this idea and introduce the idea of bit patches. The key idea is that identifiers could be sparsely distributed over a large window, such that having multiple separate bitmaps might be more space-efficient. Bitmaps of variable length could also add to efficiency. We call these bit patches, consisting of consecutive bitmaps each of fixed bit-length n . In this scheme, however, each separate bit patch would need a reference number to specify its position in the window, and a counter specifying its length in terms of number of bitmaps. For simplicity, we use n bits to represent all these values.

Computing the best set of patches is easy. Given a sorted list of integers to compress, we consider each integer in increasing order and add them to the most

recent bit patch if the integer is near enough. For blocks of length n -bits, an integer i is considered near enough if it is within $2n$ -bits of the last bit in the bit patch. Let r represent the integer represented by the last bit in the current bit patch. Then we add i to the bit patch if $i - r \leq 2n$, possibly extending it the bit patch by up to 2 units in length. Otherwise, we start a new bit patch. Once we have grouped all integers into bit patches, we can determine if a patch should be encoded using bitmaps. If the number of integers in the patch is at least 2 more than the length of the patch, it is more efficient to code it as a bitmap. Otherwise, a list form is more efficient.

This greedy linear time algorithm can be shown to give the optimal compression in our patch representation. On average, using bit patches gives us about a 15x compression over a regular list of integers, and adds the flexibility of variable sizes to bitmaps.

5.2.3 Distributed State Agreement

With sequence numbering and its exchange as primitives, we can have two neighboring nodes agree on some state. This is necessary because sending a packet is not instantaneous. In the delay, packets in response to the sender might have left the receiver. Knowing the state the receiver was in when it sent the packet allows the sender to make better decisions for its next transmission.

To achieve such agreement, each node simply needs to track its state at various points in time (identified by the sequence numbers of their packets), recording history in a double-ended queue or deque. They also need to incorporate the last heard sequence number from their neighbors into every packet they send. This way, once a node receives this “echo” of their own sequence number from a neighbor, it can affirmatively conclude that the data from that neighbor will reflect information that is consistent with the recorded state dated with that sequence number.

We use this to achieve several things in our protocol. Firstly, clients can efficiently inform their servers of the coded packets they have received, apart from sending sequence numbers in bit patches. Once clients hear back from their server, they can determine the sequence numbers which the server already knows about, and remove

them from future transmissions. Secondly, we have all nodes broadcast to neighbors the unexpired native packet identifiers they have in memory. These are similar to reception reports in COPE, except we use bit patches. Distributed state agreement allows us to keep our patches small.

Finally, we also use this to determine when to actually purge a native packet's data from memory. We note in a multi-hop network with multiple flows, even if a node has deemed that a packet has expired and thus no longer transmits it or codes it for future transmissions, it cannot delete the packet immediately. This packet might still exist in neighboring nodes where it has not expired or is sitting in the queue as part of a coded reception, and so this node might still need the data to decode these packets. Thus, nodes only delete packets permanently once they achieve agreement with neighbors on the earliest unexpired packet.

5.3 Protocol Implementation

5.3.1 Packet Header

FLOSS adds a variable-length header to packets, as shown in Figure 5-1. This header is situated between the MAC header and any encoded data of the packet, and contains 3 required fields: the packet sender's IP address, and 2 bit flags to denote the presence of two optional portions (the data portion and the response portion). The required fields are shown shaded in the figure.

The response portion, if present, contains flow metadata and server metadata. The flow metadata include, for each flow, the earliest unexpired packet at the node, the latest received packet, and the latest packet it knows of, so that its servers and clients can perform appropriate packet expiry. The server metadata in the response portion contains a sequence number bit patch for each server, which are used by the node to acknowledge to its servers the sequence numbers that have been received.

The data portion, if present, contains a sequence number, client metadata and the vector of coefficients for the coded data. For each client of the node, the header

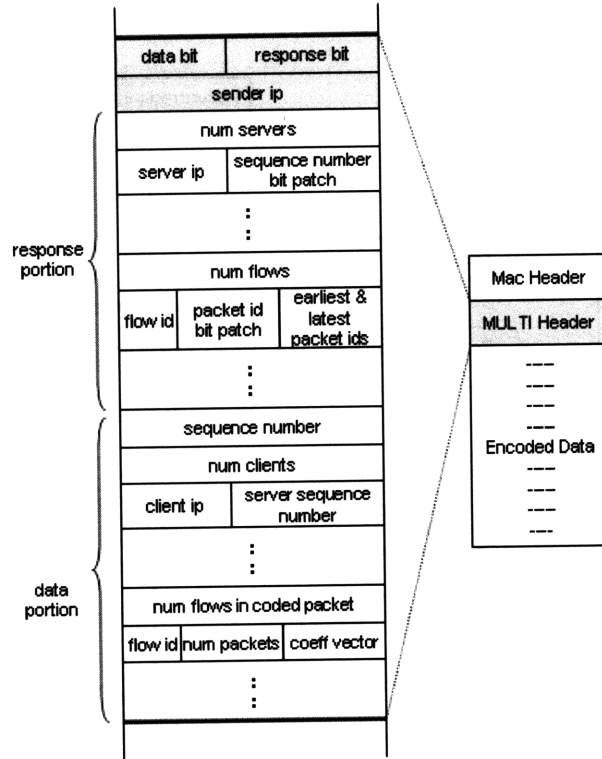
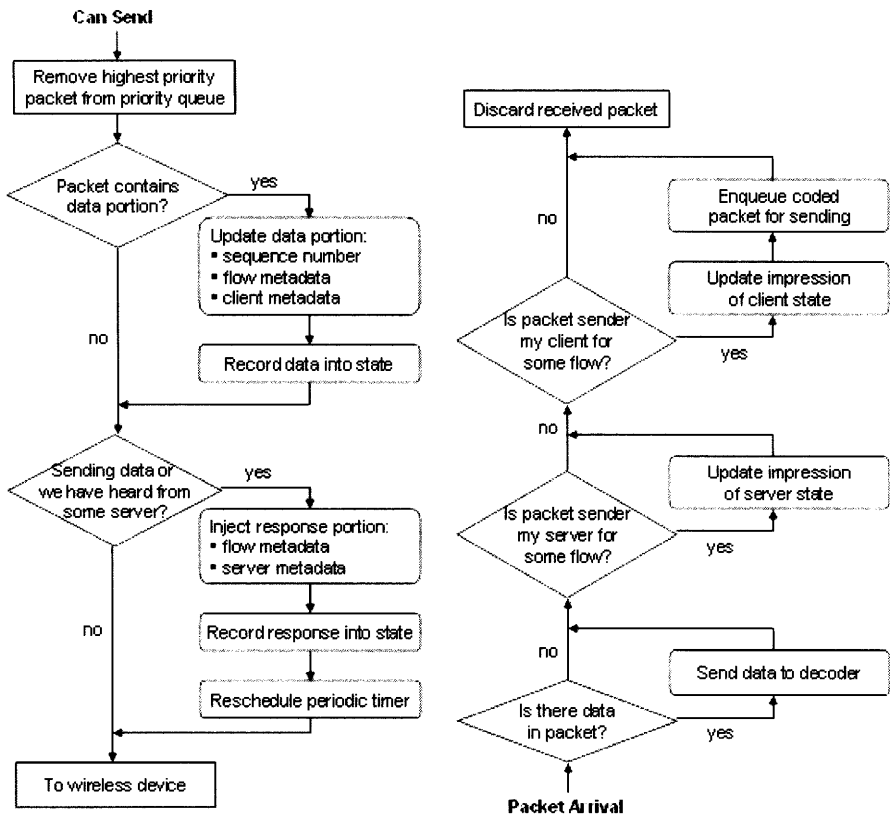


Figure 5-1: Packet header format for FLOSS. Required fields are shown shaded.

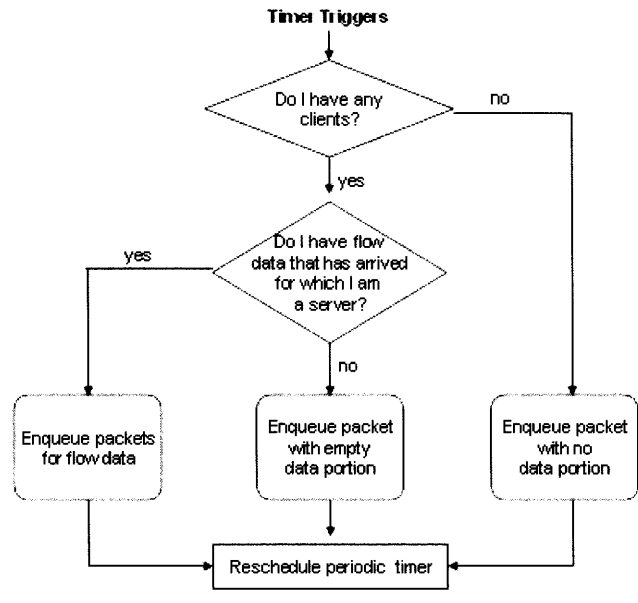
indicates the most recent server sequence number “echo” it has heard from each client. The coefficient vector is represented compactly by sorting them into flows. For each flow, we list the number of packets from that flow that are coded together, followed by a $(packet\ id, coefficient)$ tuple for each packet from that flow.

Our compact representation of the header allows us to keep the overhead small. On average, our header adds several tens of bytes to the packet length. In our current implementation, header length grows linearly with the number of clients, servers and flows that go through each node. It might be possible to reduce this by only inserting a subset of all the header information in each transmission, and have receivers deal with possibly missing information as if it got lost. However, this is something that we do not investigate in this thesis.



(a) Sender side

(b) Receiver side



(c) Periodic timer

Figure 5-2: **Architectural overview and control flow within FLOSS.** We show the control flow for the sender and receiver sides of our protocol, and that of a periodic timer at each node.

5.3.2 Control Flow

Figure 5-2 shows the architecture of the FLOSS protocol. These 3 control flows gives an overview of the decision processes that runs on each node in our network.

On the sending side (Fig 5-2(a)), packets are pre-coded and placed in a priority queue beforehand, keyed on the expiry timestamp of the first packet. Pre-coding of packets reduces delay in sending out a coded packet, while the priority queue ensures that nodes send packets in order of expiry time. The control flow for the sending is triggered when the MAC signals a sending opportunity. At this instance, the node removes the highest priority packet and checks if it contains a data portion. If so, it is sending out a coded packet. The node updates the data portion with an incremented sequence number and adds metadata about flows and clients. The node also records some state for this sequence number in its history. Next, the node checks if it is sending a packet with a data portion, or if it has received some data from at least one of its servers. If either of these conditions is satisfied, the node needs to inject a response portion into the packet. It records this response into its historical state, and reschedules the timer.

On the receiving side (Fig 5-2(b)), upon the arrival of a packet, the node first checks to see if there is data in the packet. If so, it sends the data containing coded packets to the decoder. The decoder tries to extract useful data from this packet, or stores it for later if there are still missing dependencies. Decoded packets are sent to the application, and kept in memory until it expires. Next, the node checks to see if the packet came from one of its servers for some flow. If so, it uses the contained response portion to update its impression of the server state. The same procedure is repeated if the packet came from one of its clients. This time, client state is updated, and more coded packets are scheduled for transmission based on the latest state.

The timer (Fig 5-2(c)) is scheduled to fire periodically after every `response_period` milliseconds, but it can be rescheduled to fire later whenever a packet is sent out. When the trigger does fire, it means that no packet has been transmitted from this node for the past response period. The node proceeds to enqueue a control packet.

First, it checks if it has any downstream clients. If it does not, it enqueues a packet without a data portion. This would get injected with the response portion which needs to be sent out. If the node has at least one client, it checks to see if any flow data for any clients has arrived and has not yet been sent out after a whole response period. If this is true, it means the node has not received any responses from any clients for a while, probably due to congestion or losses. Nevertheless, this node's timer has fired and it will enqueue the data for sending.

5.3.3 Playout Buffering

We modify VLC to incorporate a jitter buffer for playback. This buffer holds and reorders packets as they arrive for a fixed period of `jitter.buffer` ms before delivering them to the video decoder, allowing enough time for retransmissions of lost packets to take place.

5.4 Coding Algorithm

5.4.1 Packet Selection

At the core of FLOSS is the packet coding algorithm, shown in Algorithm 1. Upon receiving a response from one of its clients, a server node updates its impression of the client's state, and determines the native packets that are missing from that client. These packets are queued for transmission. With each queued packet, we go through the remaining clients in a random order and try to code in additional native packets. These clients could be associated with this server for different flows, but all have the need to get data packets from the server for some flow. Initially, the coded packet is just a single native packet. For each client considered, we consider if the coded packet is innovative for that client. If the client has not yet received one of the native packets coded together, then it is innovative. If not, the algorithm adds a random packet not yet received by the client to the coded packet. This packet should not make the coded packet undecodable by previously considered clients, and this needs

to be checked. To do so efficiently, we keep a set of packets that has been received by all previously considered clients, and each time, we choose from this set a new packet that has not been received by the current client.

We use linear coding with random selection of coefficients over a finite field (also known as a Galois field), so that our codes will be closer to achieving capacity for multicast than plain XOR. Also, because our coded packets will tend to have a small degree, having random coefficients would allow us to reduce redundancy in coding.

Algorithm 1 Packet Coding Procedure

```

unexpired_packets = get_received_unexpired_packets()
packet_list = determine_missing_packets(client)
received_all = get_received_packets(client)
for packet  $p$  in packet_list do
  coded_set = { $p$ }
  for client  $c$  in remaining clients do
    received_client = get_received_packets( $c$ )
    if  $c$  has everything in  $p$  then
      not_received = received_all  $\cap$  (unexpired_packets  $\setminus$  received_client)
      if not_received  $\neq \emptyset$  then
         $r$  = random packet in not_received
        coded_set = coded_set  $\cup$   $r$ 
      end if
    end if
    received_all = received_all  $\cap$  received_client
  end for
  while |coded_set| < code_width and received_all  $\neq \emptyset$  do
     $r$  = random packet in received_all
    coded_set = coded_set  $\cup$   $r$ 
    received_all = received_all  $\setminus$   $r$ 
  end while
  enqueue coded packet containing coded_set
end for

```

5.4.2 Modified Gauss-Jordan Elimination

We use a modified Gauss-Jordan row elimination algorithm (Algorithm 2) for packet decoding. Our implementation of this algorithm supports native packets from multiple flows coded together, without necessarily having a notion of a fixed batch size. We

keep coded packets in reduced row-echelon form, with the highest packet id of a coded packet having a normalized coefficient of 1. Thus, coded packets are manipulated to eliminate coefficients in other packets to achieve this form.

		Increasing packet IDs →					
		Packet	1	2	3	4	5
1. Initial state: All undecoded packets are sorted in order of increasing highest ID. Packet A is an existing packet, while B and C are new.	A:		a_2	a_3	1		
	B:	b_1	b_2		b_4	b_5	
	C:			c_3	c_4	c_5	
2. Forward elimination: we use packet A to eliminate one coefficient in B and C, and likewise use B to eliminate one in C, and we normalize.	A:		a_2	a_3	1		
	B:	b'_1	b'_2	b'_3		1	
	C:	c'_1	c'_2	1			
3. Backward elimination: we use packet C to eliminate one coefficient in packets A and B.	A:	a''_1	a''_2		1		
	B:	b''_1	b''_2			1	
	C:	c''_1	c''_2	1			

Figure 5-3: **An example of our modified Gauss-Jordan elimination for packet decoding in FLOSS.** Our algorithm allows us to depart from the notion of batches and deal with packets coded over arbitrary windows. In the above example, after the elimination is complete, we can deduce that the server just needs to send native packets 1 and 2 for the client to decode all 5 native packets.

We show an example of our algorithm in action in Figure 5-3. When we perform our elimination, we sort the coded packets in increasing order of highest packet id, as opposed to the possibly more natural representation of a decreasing order, so that it appears as if we are doing backward elimination before forward elimination. The reasoning is as follows: When new packets arrive, they are likely to have higher packet ids, putting them at the bottom of the sorted list. Thus, we keep as many of the previously normalized rows at the front of the list, and use them for elimination of the newly arrived packets. When it is time to do the reverse operation, most of the coefficients would have been eliminated from these new packets, decreasing the time needed for normalizing the row, which we do at the end only once for each new row. At the same time, our method does not require us to move any rows around. Conversely, if we were to use a decreasing order, new packets could

end up eliminating the normalized coefficients of some rows, forcing us to redo our computation of normalization for existing rows.

Algorithm 2 Modified Algorithm for Gauss-Jordan Elimination

```
packet_list = get_undecoded_packets(client)
for coded packet p in packet_list do
    remove any decoded native packets from p
    normalize coefficients of p
end for
sort packet_list by lowest packet id
for coded packet p in packet_list do
    if p has no expired native packet then
        break
    end if
    for coded packet q in packet_list before p do
        eliminate lowest packet id of p from q
    end for
    discard p
end for
sort packet_list by highest packet id
for coded packet p in packet_list do
    for coded packet q in packet_list before p do
        eliminate highest packet id of q from p
    end for
    normalize coefficients of p
    for coded packet q in packet_list before p do
        eliminate highest packet id of p from q
    end for
end for
```

Clients acknowledge to servers the sequence numbers of packets they have received from the server. This allows the server to maintain in its state the set of coded packet coefficients that have not yet been decoded. Then, when a server needs to determine the missing native packets at a client, the server looks at its impression of the client's received native packets and undecoded packet coefficients, plus what the server has sent out and what is in the queue. The server then performs elimination on these collected coefficients, to determine what will eventually be received by the client if nothing gets lost. The remaining packets which have not been received yet would be the set of missing native packets at the client.

However, notice that any remaining coded packets still provide useful information for the client, if the required packets arrive. In other words, once a client has received or decoded all the other packets that were coded together with a native packet h , it can decode the coded packet to obtain h . Thus, if we get the lower packet ids across, the highest packet id of a coded packet will now be decodable by the client. Hence, for each coded packet, the server optimistically removes from the set of missing packets the highest packet id of that coded packet, unless one native packet id of that coded packet has already been removed.

Saving the highest packet id of a coded packet for decoding last (as opposed to lowest packet id) is a natural choice. We get more opportunities to decode this packet, and the server can also focus on sending lower packet ids that are closer to expiry. In Figure 5-3, after row elimination is complete, we can deduce that the server only needs to transmit packets 1 and 2, before the client can decode all 5 packets.

The computational overhead from performing this elimination algorithm is small, because we only work on code coefficients on the server side. At the client, this elimination is done only once for data, when the client is decoding the actual packet data for each received packet.

What happens if some native packets in existing coded packets expire? Once a packet expires, the server will not choose it for coding or transmitting any more. As such, no new coded packets will arrive that would allow us to decode these coded packets with partially expired data. Fortunately, FLOSS does not just discard all these coded packets. It tries to extract as much useful data from these packets as possible. Figure 5-4 shows an example of this. Coded packets A, B and C were obtained from previous transmissions, while D is a newly received packet. Notice that native packet 1 has expired. This means 3 packets (A, B and C) contain expired data, and will be undecodable because the server will never retransmit the expired packet 1 again. However, the client can use packet A to eliminate coefficients of packet 1 in packets B and C, thus saving them from being useless. Suppose in the next time step, native packet 2 expires. Now the client can use packet B to eliminate coefficients of native packet 2 from packets C and D, saving these 2 packets again.

Packets in our scheme will expire at the same rate as native packets, thus, we can be sure that coding does not increase wastage due to expiries. FLOSS retains data in coded packets for as long as possible if they cannot yet be decoded, increasing the chance that they eventually do get decoded.

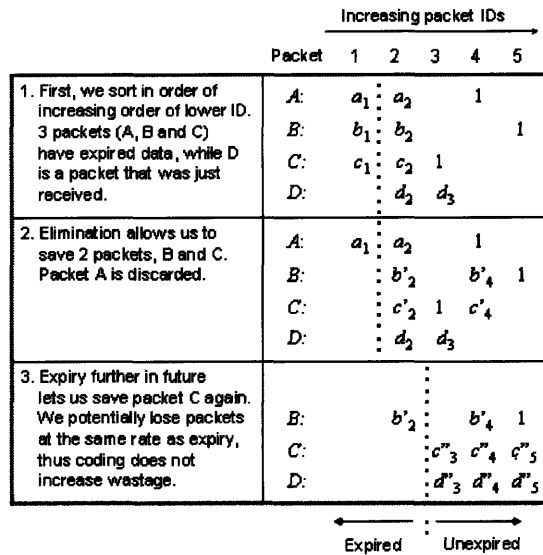


Figure 5-4: **Recovery of expired data using Gauss-Jordan elimination.** In our example above, the dashed lines represent advancing expiry times. We can use the same algorithm to extract some data from expired packets, limiting the number of coded packets that get discarded.

5.4.3 Packet Expiry Policy

FLOSS tries to minimize the memory overhead at each node. Hence, it tries to purge data the moment they are not necessary. Real-time video packets have a playback deadline, after which, the data is no longer useful. However, in FLOSS, such expired data might still be in queues awaiting transmission. FLOSS does not remove packets with expired data from the queue, nor does it try to remove the component of expired data in the coded packet, as these approaches result in delays at the sender once the medium is ready. Time was spent coding these packets, and if the node were to undo this for expired packets, the effort would be wasted. Furthermore, it might be beneficial to still send these coded packets if clients can use them to eliminate other

expired data using the above algorithm.

Thus, packet expiry just means that the native packet will not be used in any coding decisions in the future, and is separate from the actual purging of packet data. On the client side, it might be useful to keep the decoded native packet in memory to use in the decoding of packets received later, especially if servers will still send expired data. At the same time, we do not want the client to unnecessarily hold data if it will no longer be useful.

Therefore, to implement efficient packet purges, servers inform their clients of the earliest packet in their queues for each flow, or the earliest unexpired packet, whichever is earlier. This information is sent in the data portion of the packet, and is only necessary if we are sending coded native packets from that particular flow. Clients which receive this information will use it to determine their earliest unexpired packet. At the same time, through the response portion, nodes will inform their neighbors (both clients and servers) of more flow metadata. As servers, they will inform their clients of the latest packet they know of, so that clients can create appropriate records for these. As clients, they will inform their servers of the minimum of their earliest unexpired packet and the earliest packet in their queues, as well as the latest packet they have received. Their servers can then use the earliest packet information to determine if they can finally purge packet data. They can do so once all clients have sufficiently advanced their earliest packet, or if a global timer has been exceeded. The latest received packet information is useful so that servers can determine expediently if any unreceived packet is missing.

5.4.4 Incorporating Content Type Information

To give packets of different content type unequal error protection, we just need to modify the random selection procedure in Algorithm 1. We increase the probability that a packet from an I-frame is selected, relative to packets from P-frames and B-frames.

Additionally, we can model packet dependencies by having our coding procedure select a packet from a P-frame only if the corresponding I-frame has been completely

received.

These are suggestions for incorporating content type information to provide unequal error protection. However, we do not implement or evaluate these schemes in this thesis.

Chapter 6

Video Quality Assessment

In this chapter, we describe VSSIM*, a new metric for video quality assessment based on VSSIM [46]. Our metric addresses a significant limitation of the original metric and other prior work on full-reference video quality evaluation: the assumption that input videos are frame-aligned. This assumption effectively makes most metrics incapable of handling videos with skipped or repeated frames. Unfortunately, with streaming video, these display imperfections are common occurrences.

We describe our solution to this problem and show that our metric is better able to capture degradation in quality than the original metric.

6.1 Available Metrics

There have been many proposals for image [44, 45] and video [46, 33, 4, 15, 30, 47] quality metrics, as well as studies of the performance of different metrics [25, 6, 12].

In general, these metrics try to produce a metric that has a high correlation with the results of subjective evaluation by human users. These metrics account for systematic distortions like spatial shifting or temporal alignment, and use techniques like contrast masking, structure extraction, and decomposition of signals into different channels, all which emulate the human vision system.

6.1.1 VSSIM

For the purposes of evaluation of our network protocol for video streaming, we would like a full-reference video quality metric that compares the received video with the original video and determines an objective measure of its quality. Our transmission of video does not result in any global distortion such as spatial shifts or changes in gain or luminance, because our mode of transmission does not deal with analog or modified digital representations of the video. Rather, we only send packetized data of a video stream in its original encoding. Thus, the only defects that may appear are due to losses or delayed arrivals of packets.

Therefore, we decided to adopt the VSSIM [46] metric for our purposes of evaluation. Peak signal-to-noise ratio (PSNR), while commonly used as a metric for image and video quality for its computational ease, has not always been an accurate indicator of perceptual quality, as shown in [45]. Furthermore, in our case, if a frame gets transmitted and displayed with perfect quality, this gives a PSNR score of positive infinity for that frame. This would greatly skew any method of averaging the metric over all frames to get a score for the video sequence.

VSSIM, on the other hand, is based on structural similarity (SSIM) for images. Instead of looking at the visibility of error, which is what PSNR does, SSIM [45] looks at the structural distortion in images, and assigns a score ranging from 0 (worst possible distortion) to 1 (identical images). This distortion comes from 3 components, the luminance, contrast and structural differences, and is computed by analyzing the signals of the original perfect image and that of the received image. VSSIM uses the idea of SSIM, by computing the SSIM score of randomly selected 8×8 windows between pairs of frames in the two video sequences. For pairs of frames, the selected windows are each given a luminance weighting, and a weighted mean of the window scores give the score for the whole frame. The realization is that a viewer of video does not usually focus his or her attention on the darker regions of a frame, and hence windows selected there should count for less.

Additionally, once all frame scores are determined for the entire length of the

video sequence, each frame is then weighted based on the amount of motion in the frame. Again, the idea is that with large global motion, distortions in structure are not acutely visible, due to a viewer’s perceptual motion blur. The VSSIM metric is then the weighted average of the frame scores. In our implementation of VSSIM, due to the absence of motion vector information, we give all frames an equal weight in this step.

6.1.2 Limitations of VSSIM

The VSSIM metric as suggested in the original paper was used to deal with data from the VQEG [43] dataset, to evaluate processed video with visual imperfections and correlate the scores with subjective evaluation metrics. However, in our streaming experiments, our resulting video suffers from decoding imperfections due to delivery failures. These lead to decoding artifacts, skipped frames when multiple packets are missing, or repeated frames if the video stalls for rebuffering or resynchronization. As such, the frames of our received video and the original reference video are not always temporally aligned, and the VSSIM metric could be computing the structural similarity of two possibly unrelated frames. If we were to apply the original VSSIM metric to compare our collected lossy videos with the original reference videos, we cannot expect any useful indication of perceptual quality under differing loss rates, since misalignment is highly likely to occur. Furthermore, a simple detection of the average temporal shift [33] would not work, as the temporal misalignment could differ for each frame, and multiple invalid frame comparisons would affect the overall metric.

6.2 Improved Metric: VSSIM*

What we need is a way to first align the frames properly before computing the VSSIM metric. In other words, we want to compute the best possible VSSIM score over all possible frame alignments. Even if the best scoring alignment does not correspond to the true alignment, one can argue that to the human perceptual system, because of the higher VSSIM score, the perceived video would have appeared to the viewer

closest to sequence with the best alignment.

We call this metric VSSIM*, naming it after the two techniques that have inspired its creation. We describe how we can compute this metric efficiently using the A* search algorithm.

First, we formulate the problem of deciphering the best VSSIM score over all possible frame alignments as an optimization problem, one that can be solved with dynamic programming. The idea of alignment suggests similarity with the edit distance problem [23], and indeed we can use the same solution. When comparing a pair of frames from the two videos, they can either be distorted representations of each other, in which case the SSIM metric applies, or one frame is deleted or inserted. We note that, because of the nature of video playback, we will not have transpositions of frames. We associate similarity scores with deleted and inserted frames (for our experiments, we use `insert_score = delete_score = 0`), and try to minimize the “edit distance”, or in our case, maximize the similarity score between the two video sequences. We define the VSSIM* of two videos to be the maximal similarity score divided by the number of frames in the video sequence returned by the edit distance algorithm.

The concept of using edit distances for video is not new [2, 36, 39, 17], but this has not been applied with the SSIM metric for video similarity. Furthermore, we devise a faster method of computation than the traditional dynamic programming algorithm for edit distance. The computational complexity of the edit distance algorithm is $O(mn)$ for sequences of length m and n respectively, which makes it infeasible for video quality evaluation of long videos. For a one-minute video at 25fps, this translates to more than two million pairs of frames that have to be considered, and for each pair, we need to compute the SSIM score, or use VSSIM’s sampling method for greater speed. Nevertheless, this leads to n -fold increase in time complexity for evaluating a video clip with n frames, compared with a metric that does not do such alignment.

Fortunately, we can model this problem as a shortest path problem from one corner of a grid network to the opposite corner (see Figure 6-1), and use classic shortest path algorithms to solve it efficiently. In our grid, we obtain diagonal edge

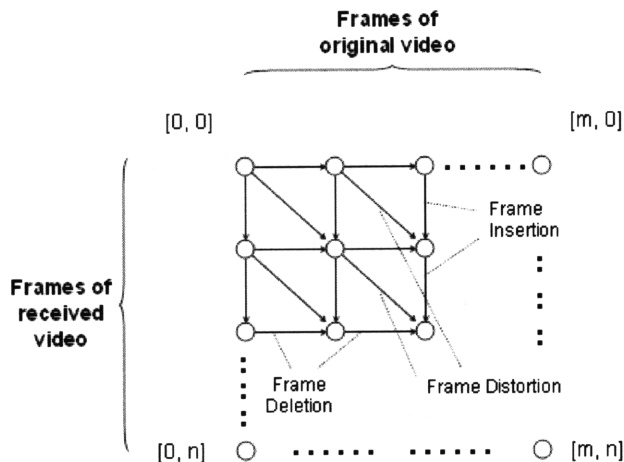


Figure 6-1: **Modeling the edit distance as a shortest path problem.** Nodes in our grid network represent pairs of frames (o_x, r_y) from the original video o and received video r respectively, while edges represent transitions between two such pairs. A diagonal edge from (o_x, r_y) to (o_{x+1}, r_{y+1}) indicates that frames o_x and r_y that are aligned but possibly suffered some distortion, and is given a weight of $1 - SSIM(o_x, r_y)$. Horizontal edges correspond to frames deleted in the received video, while vertical edges correspond to frames inserted in the received video, and these are given weights of $1 - \text{delete_score}$ and $1 - \text{insert_score}$ respectively. Finding the maximum similarity in two video sequences o and r is equivalent to finding the shortest path from (o_0, r_0) to (o_m, r_n) .

weights by subtracting the similarity scores from 1. Horizontal edges have weight $1 - \text{delete_score}$ while vertical edges have weight $1 - \text{insert_score}$. This ensures non-negative weights, and maximizing our similarity previously now becomes a problem of minimizing distance travelled. We note that we can use A* search because of a simple heuristic that is consistent and thus admissible. Let the x -coordinates denote frames of the original reference video, while y -coordinates denote frames in the observed video. Given a point \mathbf{p} on the grid, our estimate of the distance to point \mathbf{q} just assumes our computation of the SSIM along diagonal edges gives us the best possible similarity score of 1 each time. This equates to the minimal path length of zero along the diagonal stretch, leaving us with either horizontal or vertical edges remaining in the shortest possible path, if any. These edges remain if we have to delete or insert some frames respectively, and hence we consider the relevant edge weights in these cases. Our heuristic function is thus quantified by the following formula:

$$h(\mathbf{p}, \mathbf{q}) = \begin{cases} (|p_x - q_x| - |p_y - q_y|) \times (1 - \text{delete_score}) & \text{if } |p_x - q_x| \geq |p_y - q_y| \\ (|p_y - q_y| - |p_x - q_x|) \times (1 - \text{insert_score}) & \text{if } |p_x - q_x| < |p_y - q_y| \end{cases}$$

This heuristic function is simply just the lower bound on any possible path from \mathbf{p} to \mathbf{q} without knowing what the SSIM scores are for any frame. After all, our use of the heuristic is to minimize the number of pairs of frames we need to evaluate. However, we can do better if we perform some actual SSIM computation and incorporate the results into our heuristic. For example, before beginning our A* search, we can precompute SSIM scores for several horizontal and vertical strips of diagonal edges in our grid, and build a range-minimum query (RMQ) data structure [13] over each strip. Then, a heuristic query of \mathbf{p} to \mathbf{q} is likely to cross one or more such strips. We use the RMQ data structure to pick the minimum diagonal SSIM score, and pick the maximum over all strips, and add that to our heuristic. This has the effect of funnelling our paths through “valleys” in our grid network, and could reduce overall computation time. While we do not need this feature in our current evaluation, this is a possible improvement to consider for longer videos to keep the number of expanded nodes close to $O(n)$.

6.3 Further Optimizations

In practice, our use of A* gives us a several-fold speedup over the classic DP solution, with a range of 30%-70% of the nodes in the graph being expanded. However, this still yields run-times of several hours evaluation for a minute-long video. We introduce a quick step that reduces the run-time to minutes.

Before computing our metric, we first pre-process video and reduce the dimensions of each frame to a quarter of the original. This makes our image a mere $\frac{1}{16}$ of the original size. Since we are already subsampling each image when we compute the VSSIM metric, reducing the dimensions should increase the significance of each sample, albeit at the cost of a higher margin of error, since we are merging pixels

together. We find that this causes our video quality metric to deviate by about 2-5% on average. After all, from the statistics point of view, by computing similarity scores between nearly all pairs of frames, we have greatly increased the number of data samples. Reducing the dimensions introduces distortion or variance in the results, but this is mitigated by the larger number of samples we have.

This pre-processing step, however, gives us a huge speed boost of several orders of magnitude, ranging from 10x to 100x. Before, even if we needed a few samples from an image, we would need to load the whole image to memory, incurring excessive reads from disk. With a smaller image, this overhead is greatly reduced, and we can benefit more from memory caches as well. Random access of image pixels on the disk is an alternative to reading the whole image from disk, but it is unlikely to help as much. For each sample we require a square block of pixels, and random disk access could result in more disk seeks being needed.

Given that our post-experiment evaluation of the video now runs just about 4x slower than actual video playback speed, this is a worthy gain. We can thus drive the variance of our video quality metric down faster by having more streaming experiments done, as opposed to going for greater accuracy in each experimental run by not resizing our images.

6.4 Comparison of VSSIM and VSSIM*

We run several experiments to analyze the performance of VSSIM and VSSIM*. We describe each experiment and our findings below.

6.4.1 Frame Alignment

We seek to demonstrate the ability of VSSIM* to detect the proper alignment of frames.

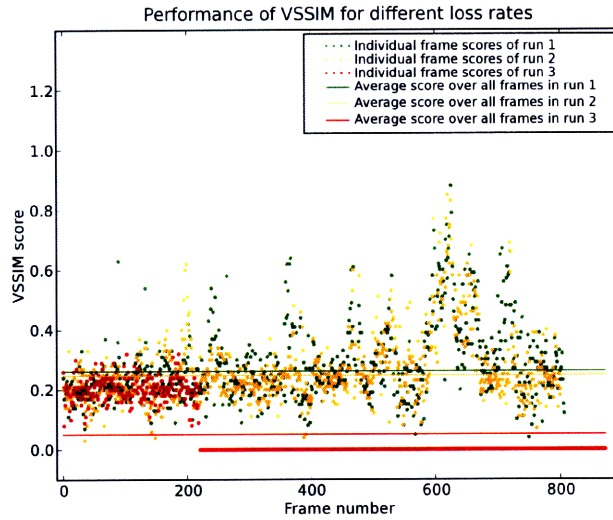
Method: We stream a 335Kb/s video three times over links of vastly differing loss rates using UDP. We then compute the original metric VSSIM and our proposed

metric VSSIM* for each experiment run. For each run, we also retrieve frame-by-frame scores of our metric and plot these over time. For VSSIM*, this equates to retrieving the shortest path, and placing a zero for each frame deleted from the original. For the regular VSSIM, we assume that a shortage of frames for comparison constitutes a frame score of zero.

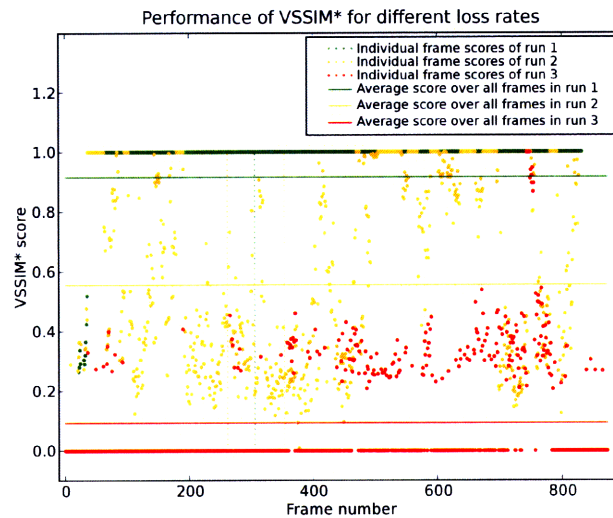
Results: The results for this experiment are summarized in Table 6.1, and our plots are shown in Figure 6-2. The x -axis represents the frame number of the original video, and the colored dots represent the frame score for the received video when compared with the respective frame of the original video. The horizontal solid lines represent the average frame score over the whole video, and is also the final score of the video. The vertical dashed lines under VSSIM* (6-2(b)) indicate positions where our metric has detected frame insertions. These frames are given scores of zero, but are not shown because the x -axis indicates the frame number of the original video only.

Our new metric is able to detect several stretches of perfect video in all 3 runs, while the original metric does not find any. The SSIM score of two frames is 1 if and only if they match exactly. Although we take samples to compute the SSIM score at each frame, at a hundred random samples per pair of frames and multiple consecutive frames having a perfect score, we can be quite certain that the stretches identified by our metric are stretches where the correct frame alignment has been found.

The plots also show that even for a video transmitted with a low loss rate and many perfectly received frames, there are inserted frames in the received video. This could be due to a slight synchronization issue between the streaming server and the client, causing the client to repeat a frame to remain synchronized with the server's display rate. Regardless of the reason for the appearance of inserted frames, our findings imply that detecting these inserted frames are necessary to find the proper alignment of the two video sequences.



(a) Regular VSSIM



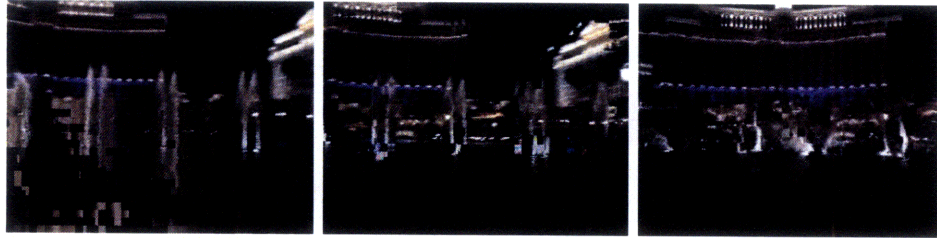
(b) Proposed VSSIM*

Figure 6-2: **Performance of video quality metric under different UDP loss rates.**

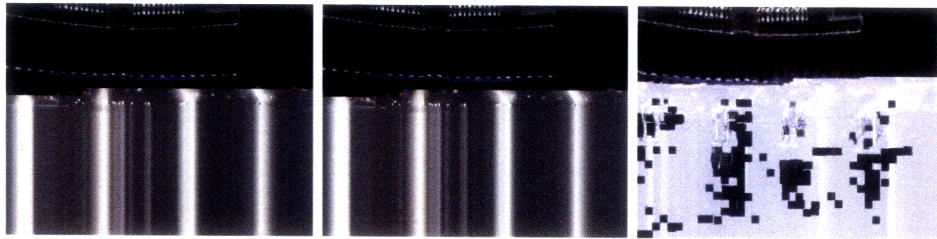
We plot the individual frame scores over all frames of the video, and use horizontal lines to indicate the average over all frames. Vertical dashed lines indicate frame insertions as computed by our metric.



(a) Run 1 (1% loss rate). The frames of this video are mostly perfect.



(b) Run 2 (53% loss rate). In these frames, we see some blockiness and decoding artifacts.



(c) Run 3 (83% loss rate). Under heavy losses, we have stalling (identical frame 57 and 67), incompletely rendered frames and unnatural colorization.

Figure 6-3: **Snapshots of observed video at different loss rates.** We show frames 57, 67, and 208 of the received video of our runs. These frames give a sense of possible manifestations of decoding error at various loss rates.

6.4.2 Indication of Quality

We wish to determine if VSSIM* is an accurate indicator of video quality, and if it is better than VSSIM.

Method: Using the results from the previous experiment, we compare 3 metrics listed in Table 6.1: the percentage of received frames, the VSSIM score, and the VSSIM* score. We use as reference snapshots of 3 representative frames from each received video in Figure 6-3. These frames depict some of the typical errors observed when watching each of the three received videos.

Runs	Loss rate	Received frames (%)	VSSIM	VSSIM*
1	1%	807/873 (92%)	0.2606	0.9146
2	53%	801/873 (92%)	0.2463	0.5539
3	83%	220/873 (25%)	0.0508	0.0928

Table 6.1: **Comparison of regular VSSIM metric and proposed VSSIM* metric.** We list for each run the overall loss rate, the number of received frames, and the computed VSSIM and VSSIM* scores.

Results: We note that the video from runs 1 and 2 have nearly similar number of received frames. However, the scores by our metric and opinions based on human observation of the videos and snapshots differ greatly. While there might be the right number of frames, blocks within the frame might be decoded poorly, or error from previous bad frames might propagate forward in time in the absence of new key frames. Hence, we can conclude that the percentage of received frames, while easy to measure, is not indicative of quality.

Similarly, the regular VSSIM gives comparable scores for runs 1 and 2. Looking at Figure 6-2, we can determine that the individual frame scores for VSSIM are scattered over the whole range from 0 to 1, giving a low average score. However, with proper frame alignment, VSSIM* has more frames with perfect scores. Run 1, having long stretches of perfect frames, naturally produces a higher average score than run 2 or 3. We find that the VSSIM* scores are closely correlated with the loss rate.

From Figure 6-3, we can get a sense of the possible and sometimes unpredictable consequences of packet losses to video. Figure 6-3(a) shows perfect frames most of the time. In Figure 6-3(b), the medium losses cause blockiness and some decoding artifacts. However, in Figure 6-3(c), heavy losses can cause stalling (same displayed image at frame 57 and 67), incomplete rendering of a frame, and even unnatural coloring.

6.4.3 Relationship of Loss Rate and Metric Score

We test the metrics in a wider range of settings to determine the relationship between packet loss rate and the scores produced by the metrics. Ideally, there should be some

indication of a relationship between these two variables.

Method: We stream videos of differing bit rates over different pairs of nodes in our testbed using UDP, so that packet losses are irrecoverable. We repeat these experiments about a hundred times for each video, and do a scatter plot of computed video quality versus loss rate.

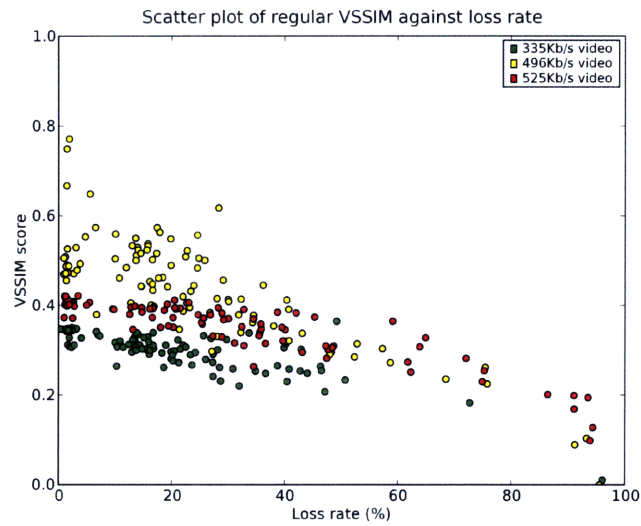
Results: Our plots are shown in Figure 6-4. Using the VSSIM* metric, we can see that a clear relationship between the loss rate of the channel and the perceived video quality. However, with the original VSSIM metric, this relationship is not as clear. In fact, the lower the loss rate, the greater the deviation in quality, which is counter-intuitive.

Our metric gives us a more accurate reflection of the quality of the links that were used to stream the videos.

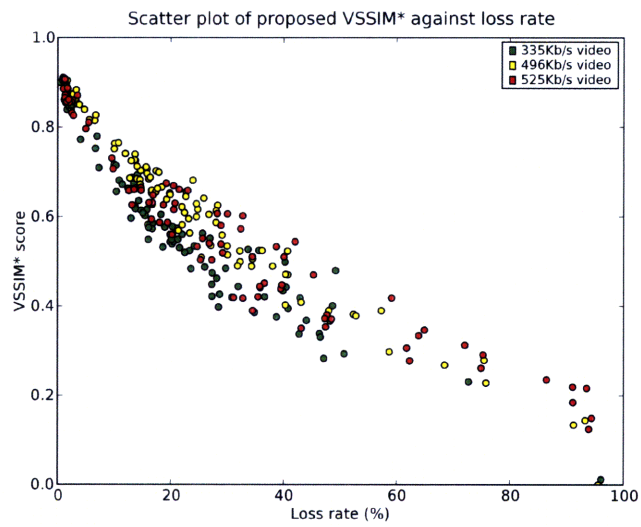
6.5 Summary

Given the above analysis, we can conclude that evaluation of video quality is a hard problem. While this thesis does not devote itself to the study of this field in detail, the formulation of a new metric VSSIM* has been helpful to our experimental evaluation of collected video samples. We note that the above technique can be extended to accommodate any kind of image quality metric that produces a bounded score, provided the metric we want for video is the best possible average score over all possible frame alignments.

For the experiments described in the following chapter, we use this proposed VSSIM* metric to measure the improvement in using our network protocol. We also investigate possible contributing factors for improved video quality such as increased throughput and reduced latency in Experiment 7.2.2.



(a) Regular VSSIM



(b) Proposed VSSIM*

Figure 6-4: **Scatter plot of video quality score against UDP loss rate.** Each dot represents a streaming experiment between two nodes, plotted at their observed loss rates and computed quality. Our proposed VSSIM* shows a more defined relationship between loss rates and video quality.

Chapter 7

Experimental Evaluation

We run a series of experiments on a mesh network of 12 nodes, evaluating the performance of the different protocols. We used common settings of `response_period = 50ms`, `jitter_buffer = 1000ms`, `code_width = 0` for our experiments. The protocols used are described here:

- FLOSS is the protocol we proposed in the earlier chapters. It performs network coding on transmitted packets, exploits opportunistic data receptions, and uses application information for packet expiry.
- BC+RETRANS works like FLOSS, except that it does not do network coding.
- BC works like BC+RETRANS, except that it only transmits packets once.
- BATCH uses fixed batch sizes of 3 packets each, and sends coded combinations of packets from a batch when requested until the batch is totally expired.

We compare the perceived video quality at the destination nodes of each flow under varying network topologies and conditions. Our experimental contribution is summarized in Table 7.1, and the experiments reveal the following results.

(a) For the Single-hop Multicast Scenario.

- With low bit-rate video, FLOSS gives comparable video quality compared with BC+RETRANS, but uses fewer transmissions. For high bit-rate video, FLOSS sustains quality up to a higher loss threshold than BC+RETRANS.

- The marginal increase in quality for each additional packet beyond the original video size is higher for FLOSS than BC+RETRANS, indicating that network coding is successful in compressing more data into a single transmission. As loss rates increase, the benefit increases.
- FLOSS gives much better quality than BATCH at high loss rates, while at low loss rates, their performance is about the same.
- Independent of loss rates, FLOSS has lower average latency than BC+RETRANS, while BATCH has the greatest per-packet latency. FLOSS gives better throughput than BATCH, and is slightly better than BC+RETRANS.

(b) For the Multiple-hop Multicast Scenario with a Single Flow.

- Opportunistic receptions and routing allows us to cut down on redundant transmissions in the network, decreasing network load. As a result, more data can get through the network and video quality improves.
- Network coding is also effective at each forwarding with more than 1 client, evidenced by the increase in video quality when network coding is performed.

(c) For the Multiple-hop Multicast Scenario with Multiple Flows.

- We see performance gains in two areas: a decreased network load and increased video quality. These gains come from two components, namely, opportunistic receptions and network coding.
- Network coding effectively reduces congestion at the routing node by decreasing the number of transmissions needed by that node. This has the positive side-effect of allowing the edge nodes to transmit more frequently and get more data to the routing node.
- For the same amount or less network load, FLOSS achieves better video quality than BC+RETRANS. The contribution to video quality by each packet improves from 5-30% through the use of network coding and opportunistic receptions.

Experiment	Section	Result
Comparison of protocols	7.2.1	FLOSS achieves higher quality video in fewer transmissions.
Components of quality gains	7.2.2	Throughput, latency and opportunism.
One-way tree topology	7.3.1	FLOSS sustains quality better.
Two-way chain topology	7.4.1	FLOSS improves packet value by 30%.
Three-way conference topology	7.4.2	FLOSS improves packet value by 5.5%.

Table 7.1: **A summary of experimental evaluations of the FLOSS protocol.** Each experiment is described in greater detail in the respective sections listed above.

7.1 Testbed Environment

(a) **Characteristics.** We use a 12-node testbed of standard Intel Celeron 2.53GHz machines with 512MB RAM. These nodes were placed at various locations on the same floor of our building, as shown in the map in Figure 7-1.

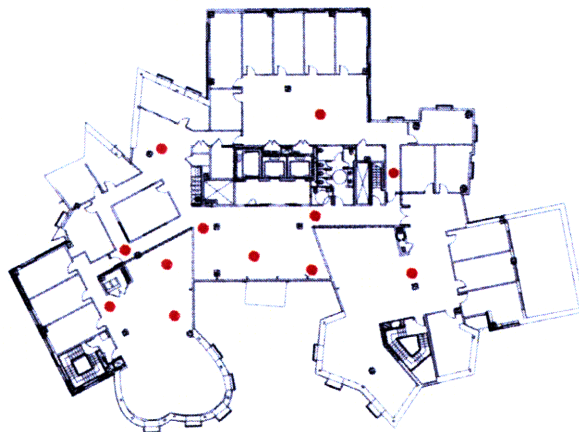


Figure 7-1: **Node locations in our testbed.** The 12 nodes of our testbed are situated on a single floor of our building.

(b) **Hardware.** Each node is equipped with a Netgear WAG311 wireless card with an omni-directional antenna. We set the transmit power level to 15 dBm, and operate in the 802.11 monitor mode. We use 802.11a channel 40 (5.2GHz) for our experiments.

(c) **Software.** Each node runs the Linux operating system. We use the Click modular router [22] and VLC [40] for our implementation.

7.2 Performance in Single-Hop Multicast

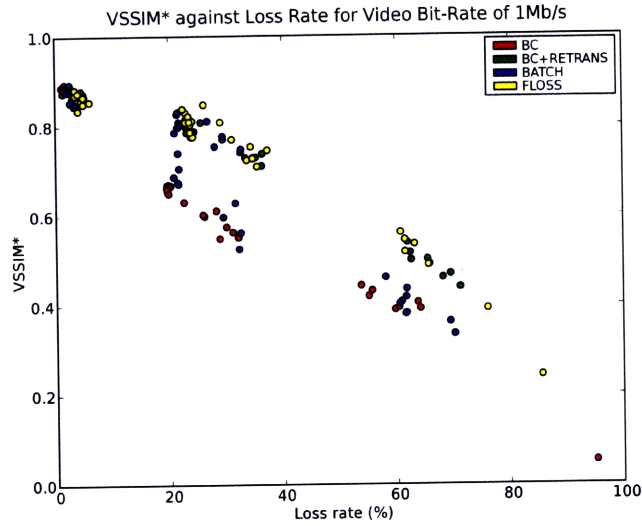
7.2.1 How Do The Protocols Compare?

Method: To answer this question, we stream two video clips to 4 clients using a transmission rate of 2Mb/s. The first clip is encoded at a bit-rate of 1Mb/s, while the second clip has bit-rate of 2Mb/s. The loss rates to these clients range from 1% to 90%, and average around 30%. We then compute the video quality for each client using VSSIM*, and make a scatter plot of the video quality against the average loss rate for each streaming session.

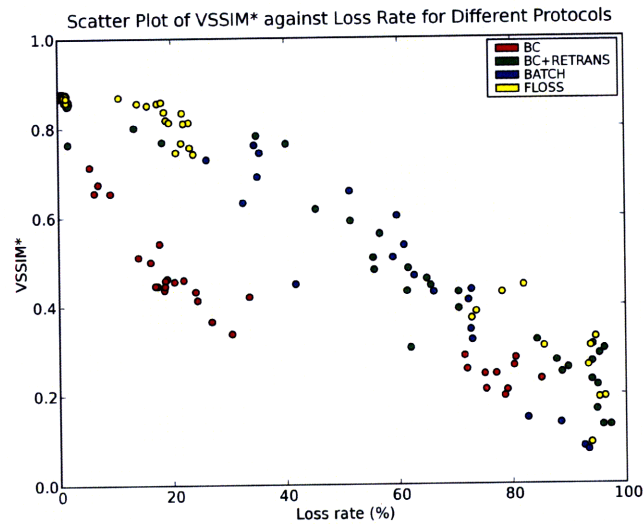
Results: Our plot in Figure 7-2 shows the performance of the different protocols for different video bit-rates. We note that for the 1Mb/s video, at low loss rates, all protocols do about the same, but as the loss rates increase, BATCH loses its effectiveness and performs only slightly better than BC. FLOSS and BC+RETRANS, however, has performance that degrades gracefully as loss rates increase. Both FLOSS and BC+RETRANS have similar performance when we consider the impact of loss rates on video quality.

However, for the higher bit-rate video, we find that FLOSS sustains high quality video up to a higher loss rate than BC+RETRANS. BATCH performs reasonably well until a loss rate of about 80%, at which point quality degrades sharply.

The plots in Figure 7-2 do not indicate the number of transmissions needed to achieve the quality of video at a particular loss rate. We would like to know is if intra-flow network coding was effective in FLOSS. We note that apart from BC, the other 3 protocols are effectively retransmissions-based protocols which correct for lost packets by retransmitting additional packets. We thus consider for each client, the number of additional packet receptions beyond BC that each of the 3 protocols FLOSS, BATCH and BC+RETRANS managed to achieve, and determine the marginal increase in video quality for each packet received. Thus, we would like a measure of how much value each retransmission added to video quality. Network coding, which allows the sending of more data with each transmission, should lead to a higher score for FLOSS.



(a) Video bit-rate of 1Mb/s



(b) Video bit-rate of 2Mb/s

Figure 7-2: **Comparison of streaming protocols for different bit-rate videos.** Each point represents a flow from a server to one client, plotted at the loss rate and video quality experienced by that client for that flow. For low bit-rate video, FLOSS and BC+RETRANS perform better than BATCH and BC. For a video of higher bit-rate, FLOSS sustains quality to a higher loss threshold than BC+RETRANS.

Our findings in Figure 7-3 confirm this. FLOSS achieves a higher marginal value

for each additional packet transmitted, for both videos of different bit-rates. That means that, for the video of lower bit-rate, for roughly the same video quality as BC+RETRANS as indicated in the Figure 7-2(a), FLOSS is using fewer packet re-transmissions. On the other hand, for the higher bit-rate video, because the medium is fully utilized, having a higher marginal value allows FLOSS to achieve a higher quality score than BC+RETRANS and BATCH.

Also, we note that the marginal value of each packet increases as the loss rates increase. This is not surprising as at higher loss rates, retransmissions play a more crucial role in contributing to video quality. BATCH however, does poorly at high loss rates, possibly doing even poorer than BC. This can be expected because at such high loss rates, it is possible that no packet in the batch gets decoded eventually.

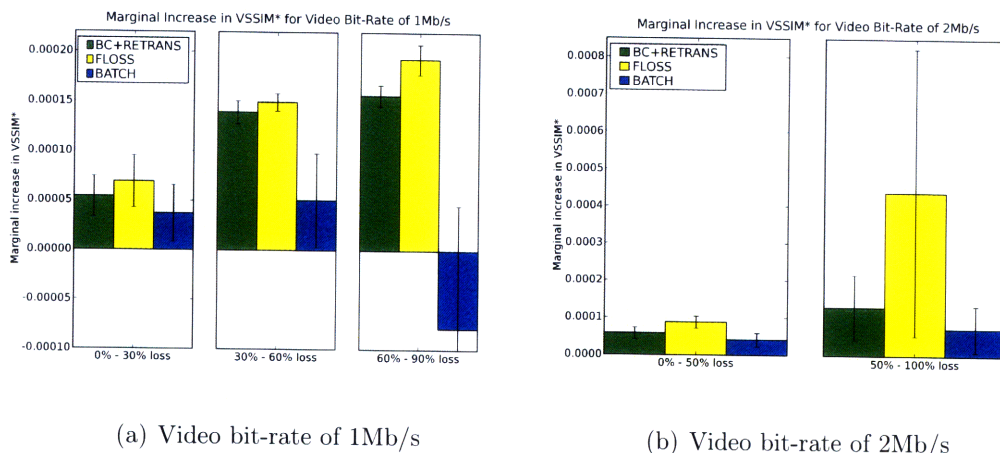


Figure 7-3: **Bar graph of marginal increase in video quality.** Each bar represents, for the 3 retransmissions-based protocols, the contribution to video quality of each received packet beyond the number of packets received in the BC protocol. Thus, we are measuring the utility of each retransmission for the 3 protocols of FLOSS, BC+RETRANS and BATCH.

7.2.2 Where Does The Improvement Come From?

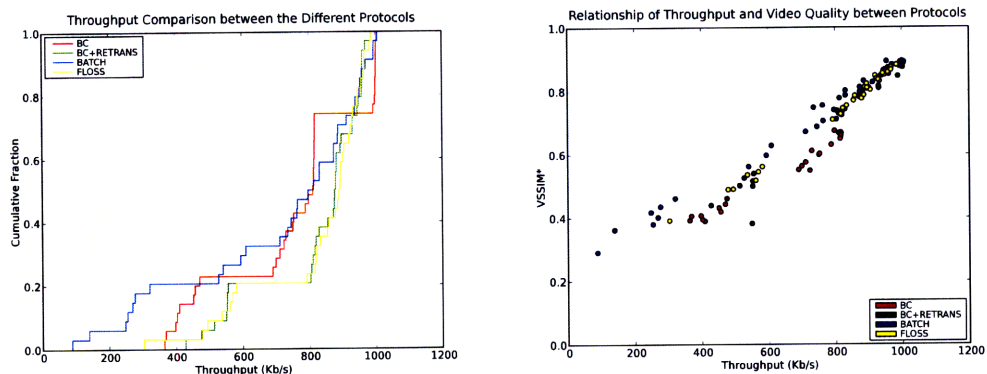
We would like to further investigate several possible contributing factors for improvement. Network coding could have led to increased throughput, and hence quality. Or

it could also reduce latency, enabling more packets to arrive on time.

(a) Increased Throughput.

Method: We take traces from the previous run of 1Mb/s video and determine the cumulative distribution function of average throughput over each run. We also do a scatter plot on average throughput and video quality for that run, to determine if there is any correlation between these.

Results: Figure 7-4 shows our analysis of throughput. We can determine that BC+RETRANS and FLOSS has the higher throughput among the 4 protocols. BATCH shows a long tail where it underperforms BC for approximately half the runs. The scatter plot indicates a linear relationship between throughput and video quality. Thus, the greater the throughput, the better we can expect video quality to be.

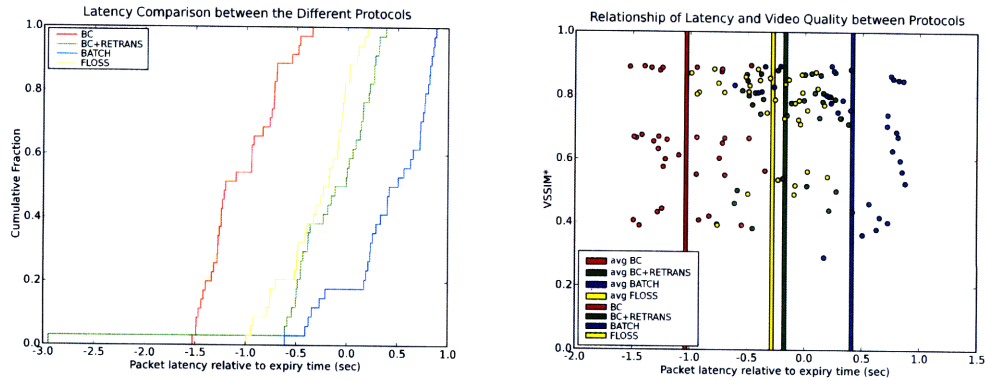


(a) Cumulative distribution of throughput. (b) Relationship between throughput and video quality.

Figure 7-4: Increased throughput improves video quality. The throughput of a streaming session is throughput of video data received by the client in that session. In the cumulative distribution of throughput over all runs of 1Mb/s video, we find that FLOSS has slightly higher median throughput than BC+RETRANS, and significantly higher throughput than the other schemes. The scatter plot shows a linear relationship between throughput and video quality, which suggests that increased throughput has a direct impact on video quality improvement.

(b) Reduced Latency.

Method: We perform similar steps for our analysis of latency, using a cumulative distribution function of difference between packet reception time and expiry time, and also use a scatter plot to infer any relationship.



(a) Cumulative distribution of latency.

(b) Absence of direct relationship between average packet latency and video quality.

Figure 7-5: **Analysis of latency for different protocols.** The vertical bars in (b) denote average packet latency over all streaming runs. FLOSS has slightly less median and average latency compared with BC+RETRANS, while BATCH has the highest average latency. There is no clear relationship between the average packet latency and video quality.

Results: We notice from Figure 7-5 that BC gives far lower latency than any other protocol. However, this happens because of the way we have implemented the jitter buffer. Our jitter buffer module in VLC pushes the expiry time of packets back by one second, in order to reorder packets that arrive out of order. Using BC, one does not require such a buffer. We note that with FLOSS and BC+RETRANS, most of the packets that get decoded are sent to the video decoder before their expiry time. However, with BATCH, this is not the case. Part of the reason is in the way we chose to implement expiry of packets and/or batches. We opted for highest delivery success, which meant that we keep a batch active until the last packet has expired. As such, this scheme keeps transmitting a batch even if some internal packets have

expired. These still get sent to the decoder after they are retrieved from the batch, even if their deadline has passed, and hence we see added latency.

(c) Opportunistic Receptions. There is another gain that we can achieve in our protocol, which comes from opportunistic receptions. In a single-hop multicast network, we do not see such gains because there is only one server transmitting to multiple clients a single hop away. We will analyze the impact of this aspect in the next two sections as we study the performance in networks with multiple hops.

7.3 Performance in Multiple-Hop Multicast

For the evaluation of our protocol in multiple-hop networks, we introduce variants of both FLOSS and BC+RETRANS that do not perform any opportunistic listening for coded data packets. We call these FLOSS/NOR and BC+RETRANS/NOR respectively, for “no opportunistic receptions”. These variants will only send and receive data along edges of the multicast tree. In each experiment, we first measure link loss rates, and build a multicast tree based on static routing using the ETX [10] metric.

Our objective is to study the gain obtained from opportunistic receptions, and differentiate between the contribution to improvement of this technique and network coding.

7.3.1 1-Way Tree Topology

Method: We evaluate our 2 normal protocols and the 2 variants on a mesh of 8 nodes. Designating one node as the source of a flow, we use static routing to obtain the shortest path to the other nodes in terms of the ETX metric. These paths form a tree as shown in Figure 7-6. We stream a 20-second 794Kb/s video clip from the source at A, with nodes C and E forwarding the data to their clients.

We look at two different metrics, the video quality perceived by each receiving node along the chain, and number of packet transmissions sent by sender along the chain.

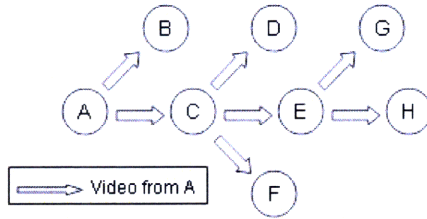
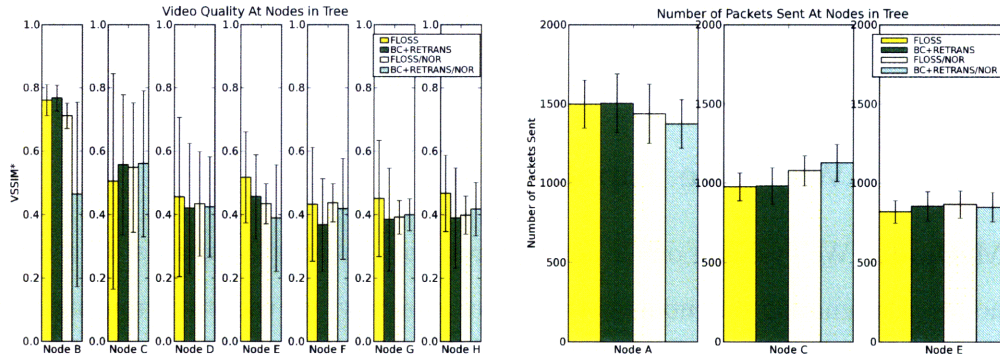


Figure 7-6: **Diagram of a tree topology.** The arrows indicate the multicast tree between 8 nodes, formed by determining static routes from the source using the ETX metric.

Results: Our plot of the perceived video quality at each node is given in Figure 7-7(a), while Figure 7-7(b) shows the number of transmissions at each of the 3 sending nodes, the first being the source, while the other 2 forwarders. We note that video quality has improved in most of the nodes, while number of transmissions has decreased slightly at forwarders primarily due to opportunistic receptions. The number of packets sent by the source, however, has increased slightly as well.



(a) Video quality at nodes in tree.

(b) Network load at transmitting nodes.

Figure 7-7: **Protocol performance on 1-way tree topology.** Video quality has improved at most nodes, while the number of transmissions has a slight decrease at forwarders primarily from opportunistic receptions.

7.4 Performance in Multiple-Hop Multicast with Multiple Flows

7.4.1 2-way Chain Topology

Method: We wish to investigate scenarios with multiple flows to determine if network coding is effective in data compression. These are scenarios where flows are bound for different destinations and cross each other via a common middle node, so that there are ample coding opportunities. Our constructed topology and two-flow multicast tree is shown in Figure 7-8. We stream two video flows with bit-rates 1.6Mb/s from either end of the chain, each destined to the nodes other than the source in the network.

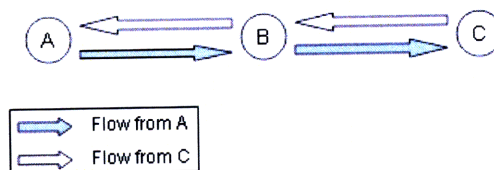


Figure 7-8: **Diagram of 2-way chain topology.** Node B sits in the middle of two opposite flows, and hence has ample opportunities for coding packets from the two flows together.

Results: Figure 7-9 shows the evaluation of our protocols on a 2-way chain of 3 nodes. We observe significant gains in video quality. This improvement comes from both opportunistic receptions and network coding separately. The gain from network coding ranges from 5-13%, while that from opportunistic receptions ranges 10-24%, for a total average gain of 13-39% in video quality score. At the same time, both these features reduce the load on the middle node. Network coding reduces this by 14-18%, while opportunistic routing decreases this by 3-7%, for a total reduction of about 20% at the average. Combining these two factors gives us a gain in per-packet quality value of each transmission of about 30% over BC+RETRANS/NOR.

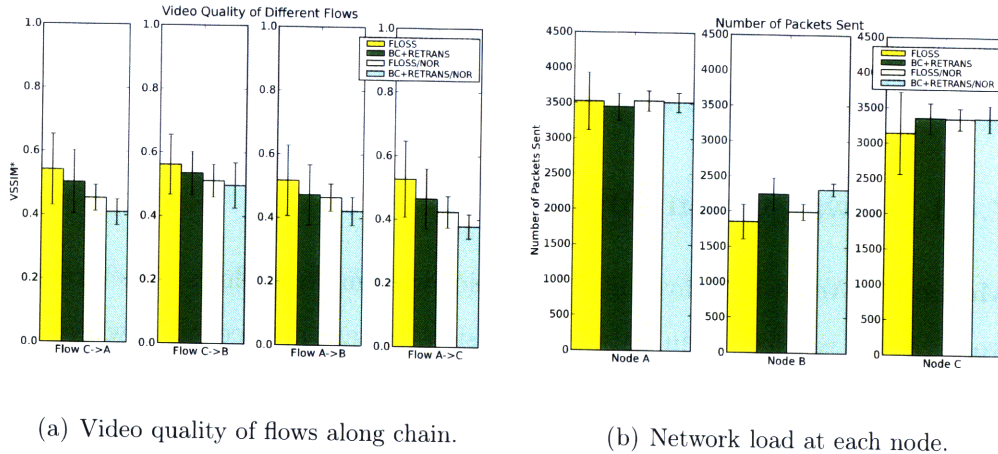


Figure 7-9: **Protocol performance on 2-way chain topology.** Both network coding and opportunistic receptions individually lead to some gain in video quality, with an average gain from 13-39%. Both of these are also responsible for decreasing the network load on the middle node by about 20%.

7.4.2 3-way Conference Topology

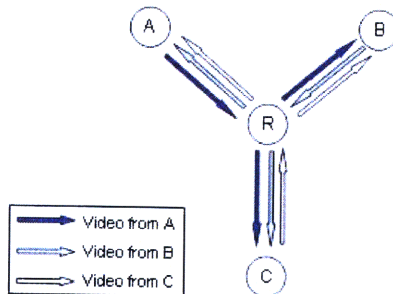
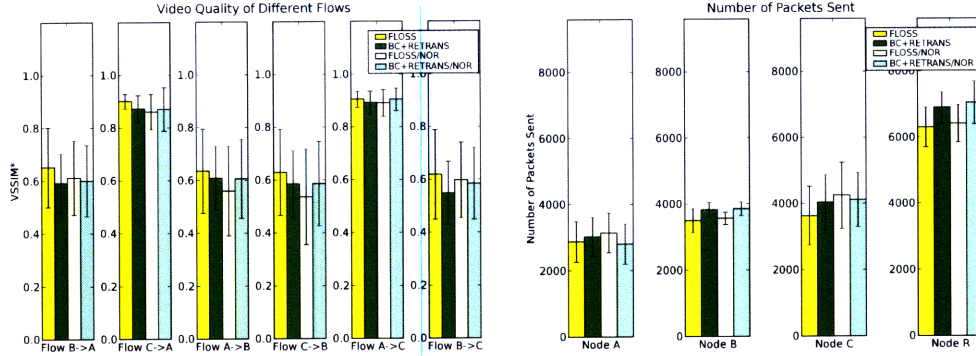


Figure 7-10: **Diagram of 3-way conference topology.** Each edge node is streaming video to the other 2 edge nodes via the middle router node. This middle node has opportunities for both inter- and intra-flow coding.

Method: We look at the 3-way video conferencing scenario, as shown in Figure 7-10. This network topology is interesting because the optimal network coding cannot be done just using inter-flow or intra-flow coding separately. Instead, one must do

both at the same time in order to achieve the capacity of the network. We stream a 34s clip of video at equal bit-rates of 496Kb/s from each of the 3 edge nodes, and have them multicast the flow with the other 2 edge nodes as destinations.



(a) Video quality of multicast flows.

(b) Network load at each node.

Figure 7-11: **Protocol performance on 3-way conference topology.** Again, network coding and opportunistic receptions both contribute to gains in video quality of about 5%, while the number of transmissions decreases by about 9% at the middle node.

Results: Figure 7-11(b) shows that our scheme improves the average video quality of all flows, while decreasing the number of transmissions each node required. Network coding contributed to an increase of 1-12% in the average video quality, while opportunistic receptions contributed 2-17%, for a total contribution of about 5%. However, at the same time, the number of transmissions at the middle node decreases by about 9%, largely due to network coding. Combining these results gives us a per-packet gain in value of 5.5%.

Chapter 8

Conclusion

8.1 Contributions

This thesis presents the first working implementation of network coding in video streaming, and experimental evaluation of such work on a testbed of commodity machines with readily available wireless cards. Furthermore, we generalize inter-flow coding of COPE to create a protocol that supports both inter-flow and intra-flow coding for unicasts and multicast of video. Our protocol applies coding in a way that keeps decoding latency to a minimum, and allows nodes in the network to benefit from opportunistic routing and reception.

We also devised a method for efficient computation of video quality for video streaming systems, which we call VSSIM*. This metric has benefits over prior work in accommodating frame insertions and deletions in video, and thus is particularly applicable to our evaluation of streaming video over lossy links.

8.2 Future Work

Future work for FLOSS would include looking at multicast subgraph selection, and determining the best multicast graph for given network conditions [27, 9]. Currently, FLOSS assumes that a multicast tree has been constructed beforehand, and performs transmission and coding along the edges of this tree. The next step would be to have

the tree constructed dynamically and in a decentralized fashion based on some cost metric, and have the link information propagated to all relevant nodes.

A natural follow-up to our coding algorithm would be to introduce a probabilistic measure of the innovation of a packet with respect to a client. In COPE, nodes make use of link loss rate information to guess if a neighboring node has received a packet, even if it has not obtained the appropriate reception report. COPE uses this information and optimistically codes packets as long as the probability of decoding is at least a certain threshold. We would like to have some optimism in FLOSS's coding, as the current scheme is overly conservative. In a similar spirit to COPE, we would like FLOSS to estimate the packet innovation to a client. Then, for any coded packet, FLOSS will try to maximize the total innovation to all clients while keeping the innovation for each client bounded close to one, perhaps at around 1.2 to allow for mistakes. This should make our network code obtain more coding opportunities and therefore perform more coding of packets.

Bibliography

- [1] R. Ahlswede, Ning Cai, S.-Y.R. Li, and R.W. Yeung. Network information flow. *Information Theory, IEEE Transactions on*, 46(4):1204–1216, Jul 2000.
- [2] Marco Bertini, Alberto Del Bimbo, and Walter Nunziati. Video clip matching using mpeg-7 descriptors and edit distance. In *CIVR*, pages 133–142, 2006.
- [3] Sanjit Biswas and Robert Morris. Opportunistic routing in multi-hop wireless networks. *SIGCOMM Comput. Commun. Rev.*, 34(1):69–74, 2004.
- [4] A. Boev, A. Gotchev, K. Egiazarian, A. Aksay, and G.B. Akar. Towards compound stereo-video quality metric: a specific encoder-based framework. *Image Analysis and Interpretation, 2006 IEEE Southwest Symposium on*, pages 218–222, 0-0 0.
- [5] M.C.O. Bogino, P. Cataldi, M. Grangetto, E. Magli, and G. Olmo. Sliding-window digital fountain codes for streaming of multimedia contents. *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 3467–3470, 27-30 May 2007.
- [6] M.H. Brill, J. Lubin, P. Costa, and J. Pearson. Accuracy and cross-calibration of video-quality metrics: new methods from atis/t1a1. *Image Processing. 2002. Proceedings. 2002 International Conference on*, 3:III–37–III–40 vol.3, 2002.
- [7] J.W. Byers, M. Luby, and M. Mitzenmacher. A digital fountain approach to asynchronous reliable multicast. *Selected Areas in Communications, IEEE Journal on*, 20(8):1528–1540, Oct 2002.
- [8] Szymon Chachulski, Michael Jennings, Sachin Katti, and Dina Katabi. Trading structure for randomness in wireless opportunistic routing. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 169–180, New York, NY, USA, 2007. ACM.
- [9] Hsu-Chen Cheng and F.Y.-S. Lin. A capacitated minimum-cost multicast routing algorithm for multirate multimedia distribution. *Intelligent Signal Processing and Communication Systems, 2004. ISPACS 2004. Proceedings of 2004 International Symposium on*, pages 211–216, 18-19 Nov. 2004.

- [10] Douglas S. J. De Couto, Daniel Aguayo, John Bicket, and Robert Morris. A high-throughput path metric for multi-hop wireless routing. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 134–146, New York, NY, USA, 2003. ACM Press.
- [11] A.G. Dimakis, Jiajun Wang, and K. Ramchandran. Unequal growth codes: Intermediate performance and unequal error protection for video streaming. *Multimedia Signal Processing, 2007. MMSP 2007. IEEE 9th Workshop on*, pages 107–110, 1-3 Oct. 2007.
- [12] U. Engelke and H.-J. Zepernick. Perceptual-based quality metrics for image and video services: A survey. *Next Generation Internet Networks, 3rd EuroNGI Conference on*, pages 190–197, 21-23 May 2007.
- [13] Johannes Fischer and Volker Heun. Theoretical and practical improvements on the rmq-problem, with applications to lca and lce. In *CPM*, pages 36–48, 2006.
- [14] C. Gkantsidis and P.R. Rodriguez. Network coding for large scale content distribution. *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, 4:2235–2245 vol. 4, 13-17 March 2005.
- [15] D. Hands, A. Bourret, and D. Bayart. Video qos enhancement using perceptual quality metrics. *BT Technology Journal*, 23(2):208–216, 2005.
- [16] T. Ho, M. Medard, R. Koetter, D.R. Karger, M. Effros, Jun Shi, and B. Leong. A random linear network coding approach to multicast. *Information Theory, IEEE Transactions on*, 52(10):4413–4430, Oct. 2006.
- [17] JunWei Hsieh, Shang-Li Yu, and Yung-Sheng Chen. Trajectory-based video retrieval by string matching. *Image Processing, 2004. ICIP '04. 2004 International Conference on*, 4:2243–2246 Vol. 4, 24-27 Oct. 2004.
- [18] Abhinav Kamra, Vishal Misra, Jon Feldman, and Dan Rubenstein. Growth codes: maximizing sensor network data persistence. *SIGCOMM Comput. Commun. Rev.*, 36(4), 2006.
- [19] Sachin Katti, Hariharan Rahul, Wenjun Hu, Dina Katabi, Muriel Médard, and Jon Crowcroft. Xors in the air: practical wireless network coding. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 243–254, New York, NY, USA, 2006. ACM Press.
- [20] Haruko Kawahigashi and Yoshiaki Terashima. Security aspects of the linear network coding. *Military Communications Conference, 2007. MILCOM 2007. IEEE*, pages 1–7, 29-31 Oct. 2007.
- [21] Ralf Koetter and Muriel Médard. An algebraic approach to network coding. *IEEE/ACM Trans. Netw.*, 11(5):782–795, 2003.

- [22] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
- [23] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, 1966.
- [24] S.-Y.R. Li, R.W. Yeung, and Ning Cai. Linear network coding. *Information Theory, IEEE Transactions on*, 49(2):371–381, Feb 2003.
- [25] Mei Hwan Loke, Ee Ping Ong, Weisi Lin, Zhongkang Lu, and Susu Yao. Comparison of video quality metrics on multimedia videos. *Image Processing, 2006 IEEE International Conference on*, pages 457–460, 8-11 Oct. 2006.
- [26] M. Luby. Lt codes. *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pages 271–280, 2002.
- [27] D.S. Lun, N. Ratnakar, M. Medard, R. Koetter, D.R. Karger, T. Ho, E. Ahmed, and F. Zhao. Minimum-cost multicast over coded packet networks. *Information Theory, IEEE Transactions on*, 52(6):2608–2623, June 2006.
- [28] A. Mahanti, D.L. Eager, M.K. Vernon, and D.J. Sundaram-Stukel. Scalable on-demand media streaming with packet loss recovery. *Networking, IEEE/ACM Transactions on*, 11(2):195–209, Apr 2003.
- [29] M. Mitzenmacher. Digital fountains: a survey and look forward. *Information Theory Workshop, 2004. IEEE*, pages 271–276, 24-29 Oct. 2004.
- [30] Samir Mohamed and Gerardo Rubino. A study of real-time packet video quality using random neural networks. *IEEE Trans. Circuits Syst. Video Techn.*, 12(12):1071–, 2002.
- [31] Bin Ni, N. Santhapuri, Zifei Zhong, and S. Nelakuditi. Routing with opportunisticly coded exchanges in wireless mesh networks. *Wireless Mesh Networks, 2006. WiMesh 2006. 2nd IEEE Workshop on*, pages 157–159, 2006.
- [32] Joon-Sang Park, Mario Gerla, Desmond S. Lun, Yunjung Yi, and Muriel Médard. Codecast: a network-coding-based ad hoc multicast protocol. *Wireless Communications, IEEE [see also IEEE Personal Communications]*, 13(5):76–81, October 2006.
- [33] M.H. Pinson and S. Wolf. A new standardized method for objectively measuring video quality. *Broadcasting, IEEE Transactions on*, 50(3):312–322, Sept. 2004.
- [34] Schulzrinne, Casner, Frederick, and Jacobson. RTP: A transport protocol for real-time applications. *Internet-Draft ietf-avt-rtp-new-01.txt (work inprogress)*, 1998.

- [35] Hulya Seferoglu and Athina Markopoulou. Opportunistic network coding for video streaming over wireless. *Packet Video 2007*, pages 191–200, 12-13 Nov. 2007.
- [36] Heng Tao Shen, Xiaofang Zhou, Zi Huang, Jie Shao, and Xiangmin Zhou. Uqlips: A real-time near-duplicate video clip detection system. In *VLDB*, pages 1374–1377, 2007.
- [37] A. Shokrollahi. Raptor codes. *Information Theory, IEEE Transactions on*, 52(6):2551–2567, June 2006.
- [38] Niveditha Sundaram, Parmesh Ramanathan, and Suman Banerjee. Multirate Media Streaming using Networking Coding. *43rd Allerton Conference on Communication, Control, and Computing, Monticello*, September 2005.
- [39] Young tae Kim and Tat-Seng Chua. Retrieval of news video using video sequence matching. *Multimedia Modelling Conference, 2005. MMM 2005. Proceedings of the 11th International*, pages 68–75, 12-14 Jan. 2005.
- [40] The VideoLAN Project. <http://www.videolan.org/>.
- [41] N. Thomos and P. Frossard. Collaborative video streaming with Raptor network coding. In *IEEE Int. Conf. on Multimedia and Expo (ICME), 2008*, Hannover, Germany.
- [42] Nikolaos Thomos and Pascal Frossard. Raptor network video coding. In *MV '07: Proceedings of the international workshop on Workshop on mobile video*, pages 19–24, New York, NY, USA, 2007. ACM.
- [43] VQEG: The Video Quality Experts Group. <http://www.vqeg.org/>.
- [44] Zhou Wang and A.C. Bovik. A universal image quality index. *Signal Processing Letters, IEEE*, 9(3):81–84, Mar 2002.
- [45] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13(4):600–612, April 2004.
- [46] Zhou Wang, Ligang Lu, and A.C. Bovik. Video quality assessment using structural distortion measurement. *Image Processing. 2002. Proceedings. 2002 International Conference on*, 3:III–65–III–68 vol.3, 2002.
- [47] A. Watson, J. Hu, and J. McGowan. Dvq: A digital video quality metric based on human vision, 2001.
- [48] Ying Zhu, Baochun Li, and Jiang Guo. Multicast with network coding in application-layer overlay networks. *Selected Areas in Communications, IEEE Journal on*, 22(1):107–120, Jan. 2004.